OMG Document bmi/2009-05-03

Proposal for:

Business Process Model and Notation (BPMN) Specification 2.0

V0.9.14

(revised submission draft)

May 22, 2009

In response to: Business Process Model and Notation RFP (OMG Document bmi/2007-06-05)

Submission Team

OMG Submitters

Axway

International Business Machines

MEGA International

Oracle

SAP AG

Unisys

Co-Authors

BizAgi

Bruce Silver Associates

IDS Scheer

Model Driven Solutions

Software AG

TIBCO Software

The following companies provided valuable review, feedback and support:

Accenture

Active Endpoints

Adaptive

Capgemini

Enterprise Agility

France Telecom

Insubria University

Intalio

Metastorm

Nortel

Red Hat Software

Vangent

Copyrights

DISCLAIMERS:

THE SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SPECIFICATION OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

You may remove these disclaimers from your modified versions of the Specification provided that you effectively disclaim all warranties and liabilities on behalf of all copyright holders in the copies of any such modified versions you distribute.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Specification or its contents without specific, written prior permission. Title to copyright in the Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Table of Contents

Table of Figures

Table of Tables

Introduction

This section presents information regarding the RFP response.

- Submitting organizations
- Supporting organizations
- Submission contacts
- Acknowledgements
- Status of this document
- Proof of Concept
- Typographical Conventions
- Guide to the submission

Submitting Organizations

The following companies are formal submitting members of OMG:

- Axway
- International Business Machines
- MEGA International
- Oracle
- SAP AG
- Unisys

Supporting Organizations

The following organizations support this specification but are not formal submitters:

- Accenture
- Adaptive
- BizAgi
- Bruce Silver Associates
- Capgemini
- Enterprise Agility
- France Telecom
- IDS Scheer
- Intalio

- Metastorm
- Model Driven Solutions
- Nortel
- Red Hat Software
- Software AG
- TIBCO Software
- Vangent

Submission Contacts

Martin Chapman, Oracle, martin.chapman@oracle.com

Dave Ings, IBM, ings@ca.ibm.com

Ivana Trickovic, SAP, ivana.trickovic@sap.com

Acknowledgements

The following persons were members of the core teams that contributed to the content specification: Anurag Aggarwal, Mike Amend, Sylvain Astier, Alistair Barros, Mariano Benitez, Conrad Bock, Martin Chapman, Rouven Day, David Frankel, Dave Ings, Pablo Irassar, Oliver Kieselbach, Matthias Kloppmann, Jana Koehler, Frank Michael Kraft, Frank Leymann, Antoine Lonjon, Sumeet Malhotra, Jeff Mischkinsky, Ralf Mueller, Karsten Ploesser, Michael Rowley, Suzette Samoojh, Vishal Saxena, Bruce Silver, Meera Srinivasan, Ivana Trickovic, Hagen Voelzer, Franz Weber, and Stephen A. White.

In addition, the following persons contributed valuable ideas and feedback that improved the content and the quality of this specification: Justin Brunt, Peter Carlson, Manoj Das, Sumeet Malhotra, Neal McWhorter, Vadim Pevzner, Pete Rivett, Jesus Sanchez, Sebastian Stein, and Prasad Yendluri.

Status of the Document

This document is an initial specification for review and comment by OMG members.

IPR and Patents

The submitters intend to contribute this work to OMG on a RF on RAND basis.

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 - 11 pt.: Standard body text

Verdana – 11 pt.: Key BPMN elements

Times/Times New Roman - 11 pt., italic: Additional BPMN elements or concepts

Helvetica/Arial - 11 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 11 pt.` Programming language elements or BPMN element attributes/model associations.

Helvetica/Arial - 11 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.

Proof of Concept

The submitters of this specification have extensive experience in building business process management tools and in implementing previous versions of the Business Process Modeling Notation specification. This specification incorporates experience the submitters have gained so far and includes proven design principles. Proof of concept implementations for this version have started and will continue in parallel with the FTF work.

Responses to RFP Requirements

See Annex A.

Guide to the Submission

The submission is organized into the following sections: Those sections are normative that are indicated as such, below.

1. Scope

The **Object Management Group** (OMG) has developed a standard **Business Process Modeling Notation** (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.

Another goal, but no less important, is to ensure that XML languages designed for the execution of business processes, such as **WSBPEL** (Web Services Business Process Execution Language), can be visualized with a business-oriented notation.

This specification represents the amalgamation of best practices within the business modeling community to define the notation and semantics of Collaboration diagrams, Process diagrams, and Choreography diagrams. The intent of BPMN is to standardize a business process modeling notation in the face of many different modeling notations and viewpoints. In doing so, BPMN will provide a simple means of communicating process information to other business users, process implementers, customers, and suppliers.

The membership of the OMG has brought forth expertise and experience with many existing notations and has sought to consolidate the best ideas from these divergent notations into a single standard notation. Examples of other notations or methodologies that were reviewed are UML Activity Diagram, UML EDOC Business Processes, IDEF, ebXML BPSS, Activity-Decision Flow (ADF) Diagram, RosettaNet, LOVeM, and Event-Process Chains (EPCs).

2. Conformance

Software may claim compliance or conformance with BPMN 2.0 if and only if the software fully matches the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. The specification defines four types of conformance namely **Process Modeling Conformance**, **Process Execution Conformance**, **BPEL Process Execution Conformance** and **Choreography Modeling Conformance**.

The implementation claiming conformance to **Process Modeling Conformance** type is not required to support **Choreography Modeling Conformance** type and vice-versa. Similarly, the implementation claiming **Process Execution Conformance** type is not required to be conformant to the **Process Modeling** and **Choreography Conformance** types.

The implementation claiming conformance to the **Process Modeling Conformance type** shall comply with all of the requirements set forth in Section 2.1. The implementation claiming conformance to the **Process Execution Conformance type** shall comply with all of the requirements set forth in Section 2.2. The implementation claiming conformance to the **BPEL Process Execution Semantics Conformance type** shall comply with all of the requirements set forth in Section 2.3.The implementation claiming conformance to the **Choreography Conformance type** shall comply with all of the requirements set forth in Section 2.4. The implementation is said to have BPMN **Complete Conformance** if it complies with all of the requirements stated in Sections 2.1, 2.2, 2.3, and 2.4.

2.1. Process Modeling Conformance

The next eight (8) sections describe **Process Modeling Conformance**.

2.1.1. BPMN Process Types

The implementations claiming **Process Modeling Conformance** must support the following BPMN packages:

- The BPMN core elements, which include those defined in the *Infrastructure*, *Foundation*, *Common*, and *Service* packages (see page 70).
- Process diagrams, which include the elements defined in the Process, Activities, *Data*, and *Human Interaction* packages (see page 153).
- Collaboration diagrams, which include Pools and Message Flow (see page 143).
- Conversation diagrams, which include Pools, Communications, and Communication Links (see page 328).

2.1.2. BPMN Process Elements

The **Process Modeling Conformance** type set consists of Collaboration and Process diagram elements, including all Task types, *embedded* Sub-Processes, CallActivity, all Gateway types, all Event types (Start, Intermediate, and End), Lane, *Participants*, Data Object (including DataInput and DataOutput), Message, Group, Text Annotation, Sequence Flow (including *conditional* and *default* flows), Message Flow, Conversations (limited to grouping Message Flow, and associating *correlations*), *Correlation*, and Association (including Compensation Association). The set also includes markers (Loop, Multi-Instance, Transaction, Compensation) for Tasks and *embedded* Sub-Processes).

Note: Implementations are not expected to support Choreography modeling elements such as Choreography Task and Choreography Sub-Process.

2.1.3. Visual Appearance

A key element of BPMN is the choice of shapes and icons used for the graphical elements identified in this specification. The intent is to create a standard visual language that all process modelers will recognize and understand. An implementation that creates and displays BPMN Process Diagrams shall use the graphical elements, shapes, and markers illustrated in this specification.

Note – There is flexibility in the size, color, line style, and text positions of the defined graphical elements, except where otherwise specified (see Page 63).

The following extensions to a BPMN Diagram are permitted:

- New markers or indicators MAY be added to the specified graphical elements. These markers or indicators could be used to highlight a specific attribute of a BPMN element or to represent a new subtype of the corresponding concept.
- A new shape representing a kind of Artifact may be added to a Diagram, but the new Artifact shape SHALL NOT conflict with the shape specified for any other BPMN element or marker.

- Graphical elements may be colored, and the coloring may have specified semantics that extend the information conveyed by the element as specified in this standard.
- The line style of a graphical element may be changed, but that change SHALL NOT conflict with any other line style required by this specification.
- An extension SHALL NOT change the specified shape of a defined graphical element or marker. (e.g., changing a square into a triangle, or changing rounded corners into squared corners, etc.).

2.1.4. Structural Conformance

An implementation that creates and displays BPMN diagrams shall conform to the specifications and restrictions with respect to the connections and other diagrammatic relationships between graphical elements. Where permitted or required connections are specified as conditional and based on attributes of the corresponding concepts, the implementation shall ensure the correspondence between the connections and the values of those attributes.

Note – In general, these connections and relationships have specified semantic interpretations, which specify interactions among the process concepts represented by the graphical elements. Conditional relationships based on attributes represent specific variations in behavior. Structural conformance therefore guarantees the correct interpretation of the diagram as a specification of process, in terms of flows of control and information. Throughout the document, structural specifications will appear in paragraphs using a special shaped bullet: Example: ♦ A TASK MAY be a target for Sequence Flow; it can have multiple *incoming* Flows. An *incoming* Flow MAY be from an alternative path and/or parallel paths.

2.1.5. Process Semantics

This specification defines many semantic concepts used in defining Processes, and associates them with graphical elements, markers, and connections. To the extent that an implementation provides an interpretation of the BPMN diagram as a semantic specification of Process, the interpretation shall be consistent with the semantic interpretation herein specified. In other words, the implementation claiming **BPMN Process Modeling Conformance** has to support the semantics surrounding the diagram elements expressed in Section 10.

Note – The implementations claiming **Process Modeling Conformance** are not expected to support the BPMN execution semantics described in Section 14.

2.1.6. Attributes and Model Associations

This specification defines a number of attributes and properties of the semantic elements represented by the graphical elements, markers, and connections. Some of these attributes are purely representational and are so marked, and some have required representations. Some attributes are specified as mandatory, but have no representation or only optional representation. And some attributes are specified as optional. For every attribute or property that is specified as mandatory, a conforming implementation SHALL provide some mechanism by which values of that attribute or property can be created and displayed. This mechanism SHALL permit the user to create or view these values for each BPMN element specified to have that attribute or property. Where a graphical representation for that attribute or property is specified as required, that graphical representation SHALL be used. Where a graphical representation for that attribute or property is specified as optional, the implementation MAY use either a graphical representation or some other mechanism. If a graphical representation is used, it SHALL be the representation specified. Where no graphical representation for that attribute or property is specified, the implementation MAY use either a graphical representation or some other mechanism. If a graphical representation is used, it SHALL NOT conflict with the specified graphical representation of any other BPMN element.

2.1.7. Extended and Optional Elements

A conforming implementation is not required to support any element or attribute that is specified herein to be non-normative or informative. In each instance in which this specification defines a feature to be "optional," it specifies whether the option is in:

- how the feature shall be displayed
- whether the feature shall be displayed
- whether the feature shall be supported

A conforming implementation is not required to support any feature whose support is specified to be optional. If an implementation supports an optional feature, it SHALL support it as specified. A conforming implementation SHALL support any "optional" feature for which the option is only in whether or how it shall be displayed.

2.1.8. Visual Interchange

One of the main goals of this specification is to provide an interchange format that can be used to exchange BPMN definitions (both domain model and diagram layout) between different tools. The implementation should support the metamodel for Process types specified in Section 13.1 to enable portability of process diagrams so that users can take business process definitions created in one vendor's environment and use them is another vendor's environment.

2.2. Process Execution Conformance

The next two (2) sections describe **Process Execution Conformance**.

2.2.1. Execution Semantics

The BPMN execution semantics have been fully formalized in this version of the specification. The tool claiming BPMN Execution Conformance type MUST fully support and interpret the operational semantics and Activity life-cycle specified in Section 14.2.2. Non-operational elements listed in Section 14 MAY be ignored by implementations claiming BPMN Execution Conformance type. Conformant implementations MUST fully support and interpret the underlying metamodel.

Note: The tool claiming **Process Execution Conformance type** is not expected to support and interpret Choreography models. The tool claiming **Process Execution Conformance type** is not expected to support **Process Modeling Conformance**.

2.2.2. Import of Process Diagrams

The tool claiming **Process Execution Conformance type** must support import of BPMN Process diagram types including its definitional Collaboration (see Table 10-1).

2.3. BPEL Process Execution Conformance

Special type of Process Execution Conformance that supports the BPMN mapping to WS-BPEL as specified in Section 15.1 can claim **BPEL Process Execution Conformance**.

Note: The tool claiming **BPEL Process Execution Conformance** must fully support **Process Execution Conformance**. The tool claiming **BPEL Process Execution Conformance** is not expected to support and interpret Choreography models. The tool claiming **BPEL Process Execution Conformance** is not expected to support **Process Modeling Conformance**.

2.4. Choreography Modeling Conformance

The next five (5) sections describe **Choreography Conformance**.

2.4.1. BPMN Choreography Types

The implementations claiming **Choreography Conformance** type must support the following BPMN packages:

- The BPMN core elements, which include those defined in the Infrastructure, Foundation, Common, and Service packages (see Chapter 70).
- Choreography diagrams, which includes the elements defined in the Choreography, and Choreography packages (see Chapter 10).
- Collaboration diagrams, which include Pools and Message Flow (see Chapter 143).

2.4.2. BPMN Choreography elements

The **Choreography Conformance** set includes Message, Choreography Task, Global Choreography Task, Choreography Sub-Process (expanded and collapsed), certain types of Start Events (e.g., None, Timer, Conditional, Signal, and Multiple), certain types of Intermediate Events (None, Message attached to Activity boundary, Timer – normal as well as attached to Activity boundary, Timer used in Event Gateways, Cancel attached to an Activity boundary, Conditional, Signal, Multiple, Link, etc) and certain types of End Events (None and Terminate), and Gateways. In addition, to enable Choreography within Collaboration it should support Pools and Message Flow.

2.4.3. Visual Appearance

An implementation that creates and displays BPMN Choreography Diagrams shall use the graphical elements, shapes and markers as specified in the BPMN specification. The use of text, color, size and lines for Choreography diagram types are listed in Section 7.4.

2.4.4. Choreography Semantics

The tool claiming **Choreography Conformance** should fully support and interpret the graphical and execution semantics surrounding Choreography diagram elements and Choreography diagram types.

2.4.5. Visual Interchange

The implementation should support import/export of Choreography diagram types and Collaboration diagram types that depict Choreography within collaboration as specified in Section 9.4 to enable portability of Choreography definitions, so that users can take BPMN definitions created in one vendor's environment and use them is another vendor's environment.

2.5. Summary of BPMN Conformance Types

Table 2-1 summarizes the requirements for BPMN Conformance.

Table 2-1 – Types of BPMN Conformance

Category	Process Modeling Conformance	Process Execution Conformance	BPEL Process Execution Conformance	Choreography Conformance
Visual representation of BPMN Diagram Types	Process diagram types and Collaboration diagram types depicting collaborations among Process diagram types.	N/A	N/A	Choreography diagram types and Collaboration diagram types depicting collaboration among Choreography diagram types.
BPMN Diagram Elements that need to be supported.	All Task types, embedded Sub-Process, Call Activity, all Event types, all Gateway types, Pool, Lane, Data Object (including DataInput and DataOutput), Message, Group, Artifacts, markers for Tasks and Sub-Processes, Sequence Flow, Associations, and Message Flow.	N/A	N/A	Message, Choreography Task, Global Choreography Task, Choreography Sub-Process (expanded and collapsed), certain types of Start, Intermediate, and End Events, Gateways, Pools and Message Flow.

Import/Export of diagram types	Yes for Process and Collaboration diagrams that depict Process within Collaboration.	Yes for Process diagrams	Yes for Process diagrams	Yes for Choreography and Collaboration diagrams depicting choreography within Collaboration.
Support for Graphical syntax and semantics	Process and Collaboration diagrams that depict Process within Collaboration.	N/A	N/A	Choreography and Collaboration diagrams depicting Choreography within Collaboration.
Support for Execution Semantics	N/A	Yes for Process diagrams	Yes for Process diagrams	Choreography execution semantics

3. Normative References

3.1. Normative

RFC-2119

- Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, IETF RFC 2119, March 1997
  http://www.ietf.org/rfc/rfc2119.txt

3.2. Non-Normative

Activity Service

- Additional Structuring Mechanism for the OTS specification, OMG, June 1999
  http://www.omg.org
- J2EE Activity Service for Extended Transactions (JSR 95), JCP
  http://www.jcp.org/jsr/detail/95.jsp

BPEL4People

- WS-BPEL Extension for People (BPEL4People) 1.0, June 2007
  http://www.active-endpoints.com/active-bpel-for-people.htm
- http://www.active-endpoints.com/active-bpel-for-people.htm
- http://www.adobe.com/devnet/livecycle/articles/bpel4people_overview.html
- http://dev2dev.bea.com/arch2arch/
- http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/
- http://www.oracle.com/technology/tech/standards/bpel4people/
- https://www.sdn.sap.com/irj/sdn/bpel4people

Business Process Definition Metamodel

- OMG, May 2008,
  http://www.omg.org/docs/dtc/08-05-07.pdf

Business Process Modeling

- Jean-Jacques Dubray, “A Novel Approach for Modeling Business Process Definitions,” 2002
  http://www.ebpml.org/ebpml2.2.doc

Business Transaction Protocol

- OASIS BTP Technical Committee, June, 2002
  http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf

Dublin Core Meta Data

- Dublin Core Metadata Element Set, Dublin Core Metadata Initiative
  http://dublincore.org/documents/dces/

ebXML BPSS

- Jean-Jacques Dubray, “A new model for ebXML BPSS Multi-party Collaborations and Web Services Choreography,” 2002
  http://www.ebpml.org/ebpml.doc

OMG UML

- Unified Modeling Language Specification V2.1.2: Superstructure, OMG, Nov 2007,
  http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF

Open Nested Transactions

- Concepts and Applications of Multilevel Transactions and Open Nested Transactions, Gerhard Weikum, Hans-J. Schek, 1992
  http://citeseer.nj.nec.com/weikum92concepts.html

RDF

- RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft
  http://www.w3.org/TR/rdf-schema/

SOAP 1.2

- SOAP Version 1.2 Part 1: Messaging Framework, W3C Working Draft
  http://www.w3.org/TR/soap12-part1/
- SOAP Version 1.2 Part21: Adjuncts, W3C Working Draft
  http://www.w3.org/TR/soap12-part2/

UDDI

- Universal Description, Discovery and Integration, Ariba, IBM and Microsoft, UDDI.org.
  http://www.uddi.org

URI

- Uniform Resource Identifiers (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter, IETF RFC 2396, August 1998
  http://www.ietf.org/rfc/rfc2396.txt

WfMC Glossary

- Workflow Management Coalition Terminology and Glossary.
  http://www.wfmc.org/standards/docs.htm

Web Services Transaction

- (WS-Transaction) 1.1, OASIS, 12 July 2007,
  http://www.oasis-open.org/committees/ws-tx/

Workflow Patterns

- Russell, N., ter Hofstede, A.H.M., van der Aalst W.M.P, & Mulyar, N. (2006). Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcentre.org
  http://www.workflowpatterns.com/

WSBPEL

- Web Services Business Process Execution Language (WSBPEL) 2.0, OASIS Standard, April 2007
  http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

WS-Coordination

- Web Services Coordination (WS-Coordination) 1.1, OASIS Standard, July 2007
  http://www.oasis-open.org/committees/ws-tx/

WSDL

- Web Services Description Language (WSDL) 2.0, W3C Proposed Recommendation, June 2007
  http://www.w3.org/TR/wsdl20/

WS-HumanTask

- Web Services Human Task (WS-HumanTask) 1.0, June 2007
  http://www.active-endpoints.com/active-bpel-for-people.htm
- http://www.adobe.com/devnet/livecycle/articles/bpel4people_overview.html
- http://dev2dev.bea.com/arch2arch/
- http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/
- http://www.oracle.com/technology/tech/standards/bpel4people/
- https://www.sdn.sap.com/irj/sdn/bpel4people

XML 1.0 (Second Edition)

- Extensible Markup Language (XML) 1.0, Second Edition, Tim Bray et al., eds., W3C, 6 October 2000
  http://www.w3.org/TR/REC-xml

XML-Namespaces

- Namespaces in XML, Tim Bray et al., eds., W3C, 14 January 1999
  http://www.w3.org/TR/REC-xml-names

XML-Schema

- XML Schema Part 1: Structures, Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, W3C, 2 May 2001
  http://www.w3.org/TR/xmlschema-1//
- XML Schema Part 2: Datatypes, Paul V. Biron and Ashok Malhotra, eds., W3C, 2 May 2001
  http://www.w3.org/TR/xmlschema-2/

XPath

- XML Path Language (XPath) 1.0, James Clark and Steve DeRose, eds., W3C, 16 November 1999
  http://www.w3.org/TR/xpath

XPDL

- Workflow Management Coalition XML Process Definition Language, version 2.0.
  http://www.wfmc.org/standards/docs.htm

4. Terms and Definitions

See Annex D - Glossary.

5. Symbols

There are no symbols defined in this specification.

6. Additional Information

6.1. Conventions

The section introduces the conventions used in this document. This includes (text) notational conventions and notations for schema components. Also included are designated namespace definitions.

6.2. Typographical and Linguistic Conventions and Style

This specification incorporates the following conventions:

- The keywords “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “MUST NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in this document are to be interpreted as described in RFC-2119.

- A **term** is a word or phrase that has a special meaning. When a term is defined, the term name is highlighted in **bold** typeface.
- A reference to another definition, section, or specification is highlighted with underlined typeface and provides a link to the relevant location in this specification.
- A reference to a graphical element is highlighted with a capitalized word and will be presented with the Verdana font (e.g., Sub-Process).
- A reference to a non-graphical element or BPMN construct is highlighted by being italicized and will be presented with the Times New Roman font (e.g., *Participant*).
- A reference to an attribute or model association will be presented with the Courier New font (e.g., `Expression`).
- A reference to a WSBPEL element, attribute, or construct is highlighted with an italic lower-case word, usually preceded by the word "WSBPEL" and will be presented with the Courier New font (e.g., WSBPEL *`pick`*).
- Non-normative examples are set off in boxes and accompanied by a brief explanation.
- XML and pseudo code is highlighted with `mono-spaced` typeface. Different font colors may be used to highlight the different components of the XML code.
- The cardinality of any content part is specified using the following operators:
  - <none> — exactly once
  - [0..1] — 0 or 1
  - [0..*] — 0 or more
  - [1..*] — 1 or more
- Attributes separated by | and grouped within { and } — alternative values
  - <value> — default value
  - <type> — the type of the attribute

6.3. Abbreviations

The following abbreviations may be used throughout this document:

This abbreviation	Refers to
WSBPEL	Web Services Business Process Execution Language (see WSBPEL). This abbreviation refers specifically to version 2.0 of the specification.
WSDL	Web Service Description Language (see WSDL). This abbreviation refers specifically to the W3C Technical Note, 15 March 2001, but is intended to support future versions of the WSDL specification

6.4. Structure of this Document

Section 7 discusses the scope of the specification and provides a summary of the elements introduced in subsequent sections of the document.

Section 8 introduces the BPMN Core that includes basic BPMN elements required for constructing various Business Processes, including collaborations, *orchestration* Processes and Choreographies.

Elements needed for modeling of Collaborations, *orchestration* Processes, Conversations, and Choreographies are introduced in sections 9, 10, 11 and 12, respectively.

Section 13 introduces the BPMN visual diagram model. Section 14 defines the execution semantics for Process *orchestrations* in BPMN 2.0. Section 15 discusses a mapping of a BPMN model to WS-BPEL that is derived by analyzing the BPMN objects and the relationships between these objects. Exchange formats and an XSLT transformation between them are provided in section 16.

7. Overview

There has been much activity in the past few years in developing web service-based XML execution languages for Business Process Management (BPM) systems. Languages such as WSBPEL provide a formal mechanism for the definition of business processes. The key element of such languages is that they are optimized for the operation and inter-operation of BPM Systems. The optimization of these languages for software operations renders them less suited for direct use by humans to design, manage, and monitor Business Processes. WSBPEL has both graph and block structures and utilizes the principles of formal mathematical models, such as pi-calculus[1]. This technical underpinning provides the foundation for business process execution to handle the complex nature of both internal and B2B interactions and take advantage of the benefits of Web services. Given the nature of WSBPEL, a complex Business Process could be organized in a potentially complex, disjointed, and unintuitive format that is handled very well by a software system (or a computer programmer), but would be hard to understand by the business analysts and managers tasked to develop, manage, and monitor the Process. Thus, there is a human level of “inter-operability” or “portability” that is not addressed by these web service-based XML execution languages.

Business people are very comfortable with visualizing Business Processes in a flow-chart format. There are thousands of business analysts studying the way companies work and defining Business Processes with simple flow charts. This creates a technical gap between the format of the initial design of Business Processes and the format of the languages, such as WSBPEL, that will execute these Business Processes. This gap needs to be bridged with a formal mechanism that maps the appropriate visualization of the Business Processes (a notation) to the appropriate execution format (a BPM execution language) for these Business Processes.

Inter-operation of Business Processes at the human level, rather than the software engine level, can be solved with standardization of the Business Process Modeling Notation (BPMN). BPMN provides a multiple diagrams, which are designed for use by the people who design and manage Business Processes. BPMN also provides a mapping to an execution language of BPM Systems (WSBPEL). Thus, BPMN would provide a standard visualization mechanism for Business Processes defined in an execution optimized business process language.

BPMN provides businesses with the capability of understanding their internal business procedures in a graphical notation and will give organizations the ability to communicate these procedures in a standard manner. Currently, there are scores of Process modeling tools and methodologies. Given that individuals will move from one company to another and that companies will merge and diverge, it is likely that business analysts are required to understand multiple representations of Business Processes—potentially different representations of the same Process as it moves through its lifecycle of development, implementation, execution, monitoring, and analysis. Therefore, a standard graphical notation will facilitate the understanding of the performance Collaborations and business *transactions* within and between the organizations. This will ensure that businesses will understand themselves and participants in their business and will enable organizations to adjust to new internal and B2B business circumstances quickly. BPMN follows the tradition of flowcharting notations for readability and flexibility. In addition, the BPMN execution semantics is fully formalized. The OMG is using the experience of the business process notations that have preceded BPMN to create the next generation notation that combines readability, flexibility, and expandability.

[1] See Milner, 1999, “Communicating and Mobile Systems: the Π-Calculus,” Cambridge University Press. ISBN 0 521 64320 1 (hc.) ISBN 0 521 65869 1 (pbk.)

BPMN will also advance the capabilities of traditional business process notations by inherently handling B2B Business Process concepts, such as *public* and *private* Processes and Choreographies, as well as advanced modeling concepts, such as *exception handling, transactions*, and *compensation*.

7.1. BPMN Scope

This specification provides a notation and model for Business Processes and an interchange format that can be used to exchange BPMN Process definitions (both domain model and diagram layout) between different tools. The goal of the specification is to enable portability of Process definitions, so that users can take Process definitions created in one vendor's environment and use them is another vendor's environment.

The BPMN 2.0 specification extends the scope and capabilities of the BPMN 1.2 in several areas:

- Formalizes the execution semantics for all BPMN elements
- Defines an extensibility mechanism for both Process model extensions and graphical extensions
- Refines Event composition and correlation
- Extends the definition of human interactions
- Defines a Choreography model

This specification also resolves known BPMN 1.2 inconsistencies and ambiguities.

BPMN is constrained to support only the concepts of modeling that are applicable to Business Processes. This means that other types of modeling done by organizations for business purposes is out of scope for BPMN. Therefore, the following are aspects that are out of the scope of this specification:

- Definition of organizational models and resources
- Modeling of functional breakdowns
- Data and information models
- Modeling of strategy
- Business rules models

Since these types of high-level modeling either directly or indirectly affects Business Processes, the relationships between BPMN and other high-level business modeling can be defined more formally as BPMN and other specifications are advanced.

While BPMN shows the flow of data (Messages), and the association of data artifacts to Activities, it is not a data flow language. In addition, operational simulation, monitoring and deployment of Business Processes are out of scope of this specification.

BPMN 2.0 may be mapped to more than one platform dependent process modeling language, e.g. WS-BPEL 2.0. This document includes a mapping of a subset of BPMN to WS-BPEL 2.0. Mappings to other emerging standards are considered to be separate efforts.

The specification utilizes other standards for defining data types, expressions and service operations. These standards are XML Schema, XPath, and WSDL, respectively.

7.1.1. Uses of BPMN

Business Process modeling is used to communicate a wide variety of information to a wide variety of audiences. BPMN is designed to cover many types of modeling and allows the creation of end-to-end Business Processes. The structural elements of BPMN allow the viewer to be able to easily differentiate between sections of a BPMN Diagram. There are three basic types of sub-models within an end-to-end BPMN model:

- Processes (Orchestration), including:
  - *Private Non-executable* (*internal*) Business Processes
  - *Private Executable* (internal) Business Processes
  - *Public* Processes
- Choreographies
- Collaborations, which may include Processes and/or Choreographies
  - A view of Conversations

Private (Internal) Business Processes

Private Business Processes are those internal to a specific organization. These Processes have been generally called workflow or BPM Processes (see Figure 10-4). Another synonym typically used in the Web services area is the *Orchestration* of services. There are two (2) types of *private* Processes: *executable* and *non-executable*. An *executable* Process is a Process that has been modeled for the purpose of being executed according to the semantics defined in Chapter 14 (see page 426). Of course, during the development cycle of the Process, there will be stages where the Process does not have enough detail to be "executable." A non-executable Process is a *private* Process that has been modeled for the purpose of documenting Process behavior at a modeler-defined level of detail. Thus, information required for execution, such as formal condition expressions are typically not included in a *non-executable* Process.

If a swimlanes-like notation is used (e.g., a Collaboration, see below) then a *private* Business Process will be contained within a single Pool. The Process flow is therefore contained within the Pool and cannot cross the boundaries of the Pool. The flow of Messages can cross the Pool boundary to show the interactions that exist between separate *private* Business Processes.

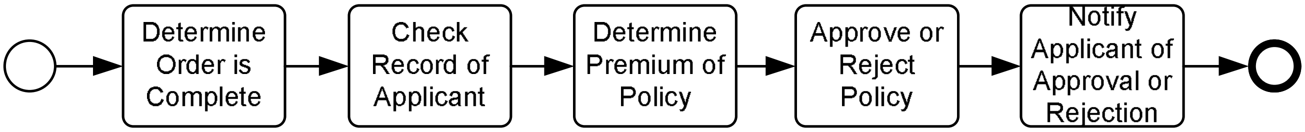

Determine Order is Complete
Check Record of Applicant
Determine Premium of Policy
Approve or Reject Policy
Notify Applicant of Approval or Rejection

Figure 7-1 – Example of a *private* Business Process

Public Processes

A *public* Process represents the interactions between a *private* Business Process and another Process or *Participant* (see Figure 10-5). Only those Activities that are used to communicate to the other *Participant(s)* are included in the *public* Process. All other "internal" Activities of the *private* Business Process are not shown in the *public* Process. Thus, the *public* Process shows to the outside world the Message Flow and the order of those Message Flow that are required to interact with that Process. *Public* Processes can be modeled separately or within a Collaboration to show the flow of Messages between the *public* Process Activities and other *Participants*. Note that the *public* type of Process was named "abstract" in BPMN 1.2.

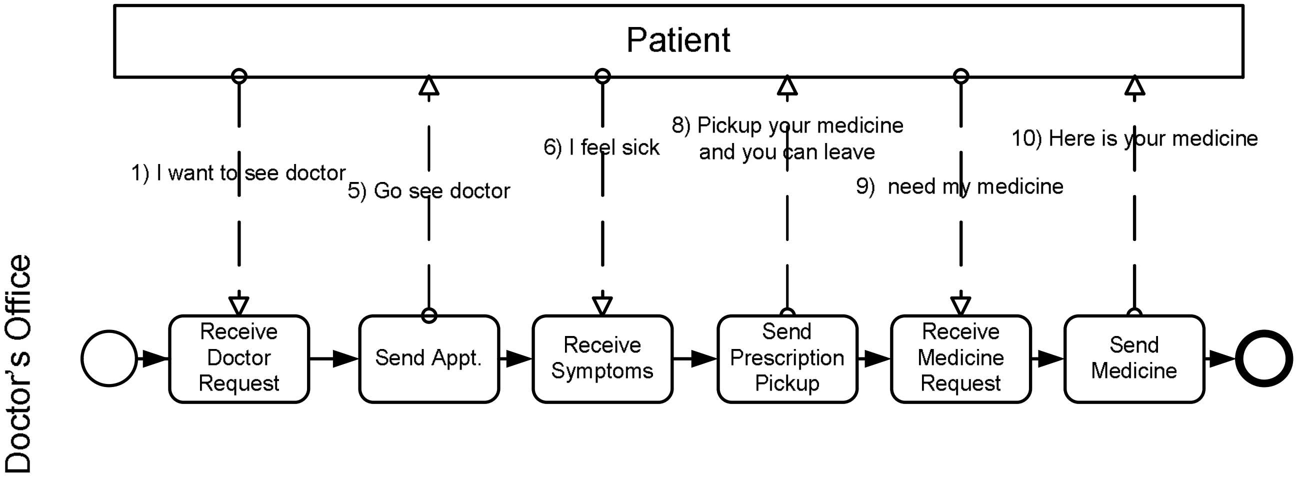


Figure 7-2 – Example of a *public* Process

Collaborations

A Collaboration depicts the interactions between two or more business entities. A Collaboration contains two (2) or more Pools, representing the *Participants* in the Collaboration. The Message exchange between the *Participants* is shown by a Message Flow that connects two (2) Pools (or the objects within the Pools). The Messages associated with the Message Flow may also be shown. The Collaboration can be shown as two or more *public* Processes communicating with each other (see Figure 7-3). With a *public* Process, the Activities for the Collaboration participants can be considered the "touch-points" between the participants. The corresponding internal (executable) Processes are likely to have much more Activity and detail than what is shown in the *public* Processes. Or a Pool may be empty, a "black box." Choreographies may be shown "in between" the Pools as they bisect the Message Flow between the Pools. All combinations of Pools, Processes, and a Choreography are allowed in a Collaboration.

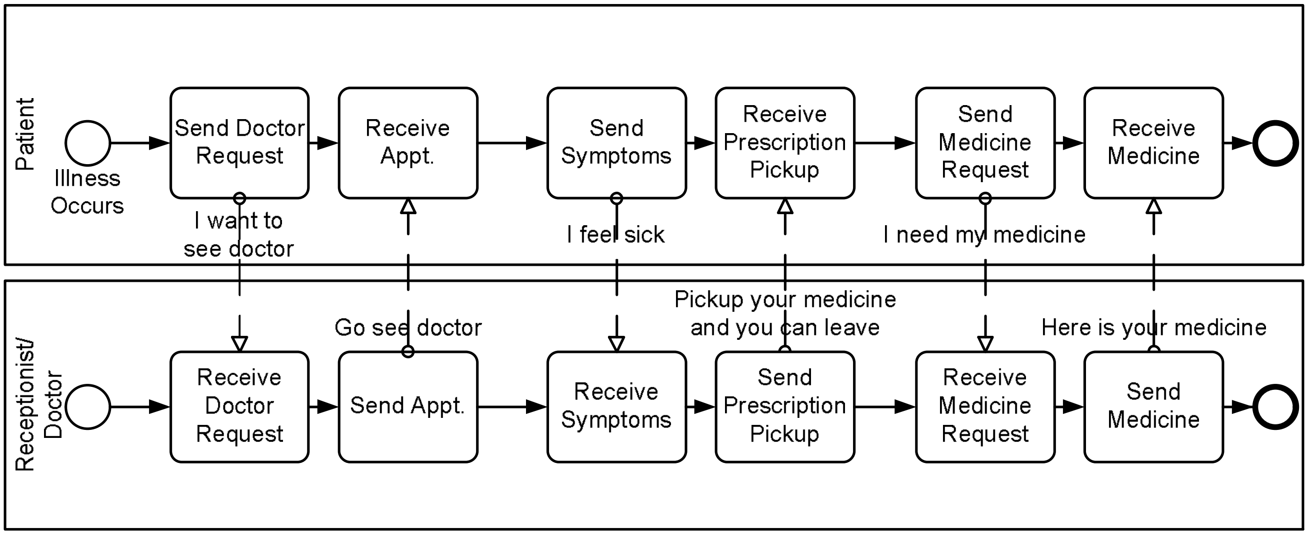


Figure 7-3 – An example of a Collaborative Process

Choreographies

A self-contained Choreography (no Pools or *Orchestration*) is a definition of the expected behavior, basically a procedural contract, between interacting *Participants*. While a normal Process exists within a Pool, a Choreography exists between Pools (or *Participants*).

The Choreography looks similar to a *private* Business Process since it consists of a network of Activities, Events, and Gateways (see Figure 7-4). However, a Choreography is different in that the Activities are interactions that represent a set (1 or more) of Message exchanges, which involves two (2) or more *Participants*. In addition, unlike a normal Process, there is no central controller, responsible entity or observer of the Process.

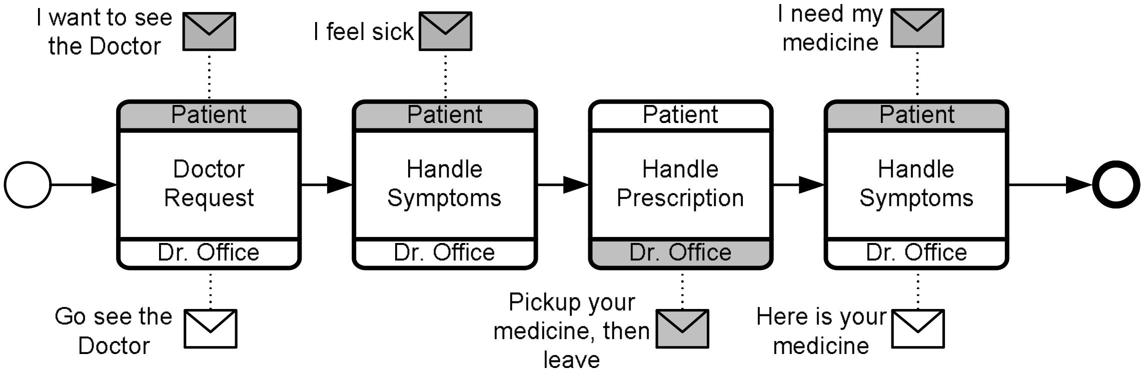


Figure 7-4 – An example of a Choreography

Conversations

The Conversation diagram is similar to a Collaboration diagram. However, the Pools of a Conversation are not allowed to contain a Process and a Choreography is not allowed to be placed in between the Pools of a Conversation diagram. A Conversation is the logical relation of Message exchanges. The logical relation, in practice, often concerns a business object(s) of interest, e.g. "Order," "Shipment and Delivery," or "Invoice."

Message exchanges are related to each other and reflect distinct business scenarios. For example, in logistics, stock replenishments involve the following types scenarios: creation of sales orders; assignment of carriers for shipments combining different sales orders; crossing customs/quarantine; processing payment and investigating exceptions. Thus, a Conversation diagram, as shown in Figure 7-5, shows Communications (as hexagons) between *Participants* (Pools). This provides a "bird's eye" perspective of the different Conversations which relate to the domain.

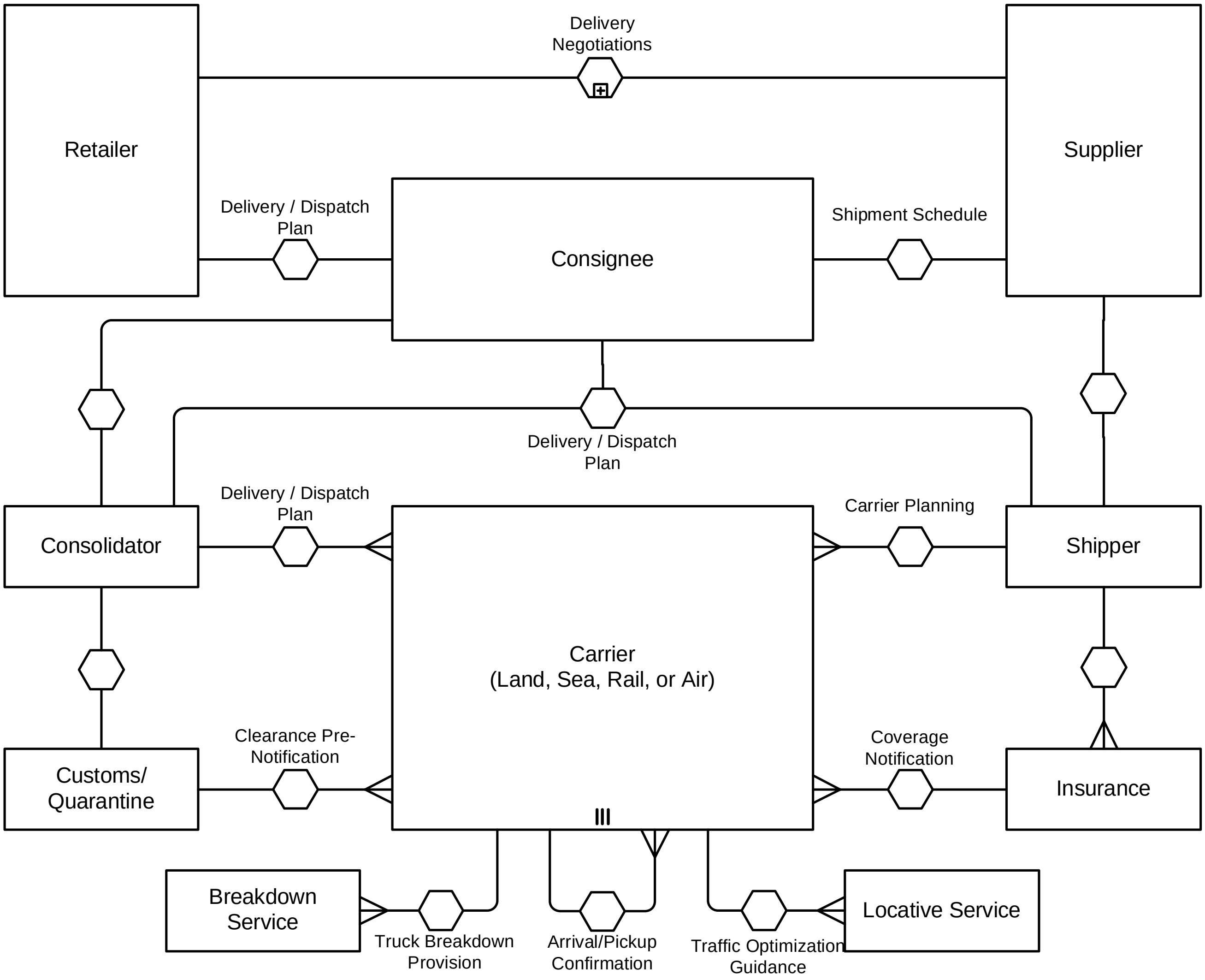


Figure 7-5 – An example of a Conversation diagram

Diagram Point of View

Since a BPMN Diagram may depict the Processes of different Participants, each Participant may view the Diagram differently. That is, the Participants have different points of view regarding how the Processes will apply to them. Some of the Activities will be internal to the Participant (meaning performed by or under control of the Participant) and other Activities will be external to the Participant. Each Participant will have a different perspective as to which are internal and external. At runtime, the difference between internal and external Activities is important in how a Participant can view the status of the Activities or trouble-shoot any problems. However, the Diagram itself remains the same. Figure 7-3, above, displays a Business Process that has two points of view. One point of view is of a Patient, the other is of the Doctor's office. The Diagram shows the Activities of both participants in the Process, but when the Process is actually being performed, each Participant will only have control over their own Activities. Although the Diagram point of view is important for a viewer of the Diagram to understand how the behavior of the Process will relate to that viewer, BPMN will not currently specify any graphical mechanisms to highlight the point of view. It is open to the modeler or modeling tool vendor to provide any visual cues to emphasize this characteristic of a Diagram.

Understanding the Behavior of Diagrams

Throughout this document, we discuss how Sequence Flow is used within a Process. To facilitate this discussion, we employ the concept of a *token* that will traverse the Sequence Flow and pass through the elements in the Process. A *token* is a theoretical concept that is used as an aid to define the behavior of a Process that is being performed. The behavior of Process elements can be defined by describing how they interact with a *token* as it "traverses" the structure of the Process. However, modeling and execution tools that implement BPMN are not required to implement any form of *token*.

A Start Event generates a *token* that must eventually be consumed at an End Event (which may be implicit if not graphically displayed). The path of *tokens* should be traceable through the network of Sequence Flow, Gateways, and Activities within a Process.

Note: A *token* does not traverse the Message Flow since it is a Message that is passed down a Message Flow (as the name implies).

7.2. BPMN Elements

It should be emphasized that one of the drivers for the development of BPMN is to create a simple and understandable mechanism for creating Business Process models, while at the same time being able to handle the complexity inherent to Business Processes. The approach taken to handle these two conflicting requirements was to organize the graphical aspects of the notation into specific categories. This provides a small set of notation categories so that the reader of a BPMN diagram can easily recognize the basic types of elements and understand the diagram. Within the basic categories of elements, additional variation and information can be added to support the requirements for complexity without dramatically changing the basic look and feel of the diagram. The five (5) basic categories of elements are:

- *Flow Objects*
- *Data*
- *Connecting Objects*
- *Swimlanes*
- Artifacts

Flow Objects are the main graphical elements to define the behavior of a Business Process. There are three (3) *Flow Objects*:

- Events
- Activities
- Gateways

Data is represented with the five (5) elements:

- Data Objects
- Data Inputs
- Data Outputs
- Data Stores
- Properties

There are four (4) ways of connecting the Flow Objects to each other or other information. There are four (4) Connecting Objects:

- Sequence Flow
- Message Flow
- Association
- Data Association

There are two (2) ways of grouping the primary modeling elements through "Swimlanes:"

- Pools
- Lanes

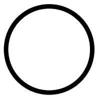
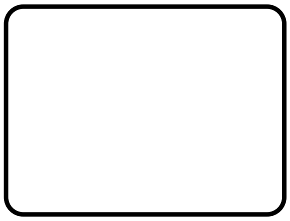
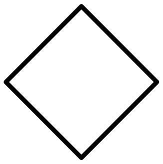
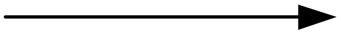
Artifacts are used to provide additional information about the Process. There are two (2) standardized Artifacts, but modelers or modeling tools are free to add as many Artifacts as required. There may be additional BPMN efforts to standardize a larger set of Artifacts for general use or for vertical markets. The current set of Artifacts includes:

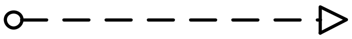
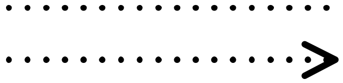
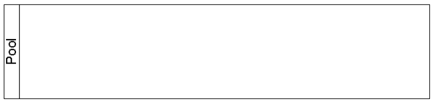
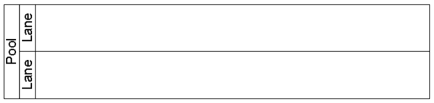
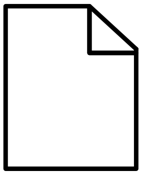
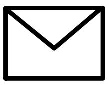
- Group
- Text Annotation

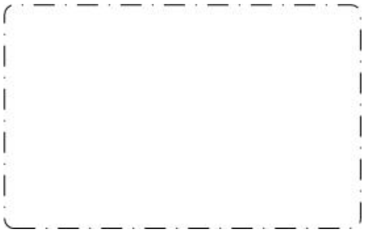
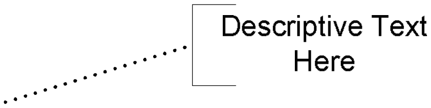
7.2.1. Basic BPMN Modeling Elements

Table 7-1 displays a list of the basic modeling elements that are depicted by the notation.

Table 7-1 - Basic Modeling Elements

Element	Description	Notation
Event	An Event is something that "happens" during the course of a Process (see page 239) or a Choreography (see page 369). These Events affect the flow of the model and usually have a cause (*trigger*) or an impact (*result*). Events are circles with open centers to allow internal markers to differentiate different *triggers* or *results*. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End.	
Activity	An Activity is a generic term for work that company performs (see page 159) in Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process Model are: Sub-Process and Task, which are rounded rectangles. Activities are used in both standard Processes and in Choreographies.	
Gateway	A Gateway is used to control the divergence and convergence of Sequence Flow in a Process (see page 295) and in a Choreography (see page 375). Thus, it will determine branching, forking, merging, and joining of paths. Internal markers will indicate the type of behavior control.	
Sequence Flow	A Sequence Flow is used to show the order that Activities will be performed in a Process (see page 129) and in a Choreography (see page 348).	

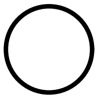
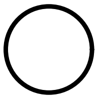
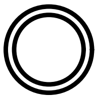
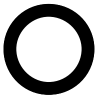
Message Flow	A Message Flow is used to show the flow of Messages between two *Participants* that are prepared to send and receive them (see page 119). In BPMN, two separate Pools in a Collaboration Diagram will represent the two *Participants* (e.g., PartnerEntities and/or PartnerRoles).	
Association	An Association is used to link information and Artifacts with BPMN graphical elements (see page 88). Text Annotations (see page 93) and other Artifacts (see page 86) can be Associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.	
Pool	A Pool is the graphical representation of a *Participant* in a Collaboration (see page 146). It is also acts as a "swimlane" and a graphical container for partitioning a set of Activities from other Pools, usually in the context of B2B situations.	
Lane	A Lane is a sub-partition within a Process, sometimes within a Pool, and will extend the entire length of the Process, either vertically or horizontally (see on page 149). Lanes are used to organize and categorize Activities.	
Data Object	Data Objects provide information about what Activities require to be performed and/or what they produce (see page 213), Data Objects can represent a singular object or a collection of objects. Data Input and Data Output provide the same information for Processes.	
Message	A Message is used to depict the contents of a communication between two *Participants* (as defined by a business PartnerRole or a business PartnerEntity—see on page 112).	

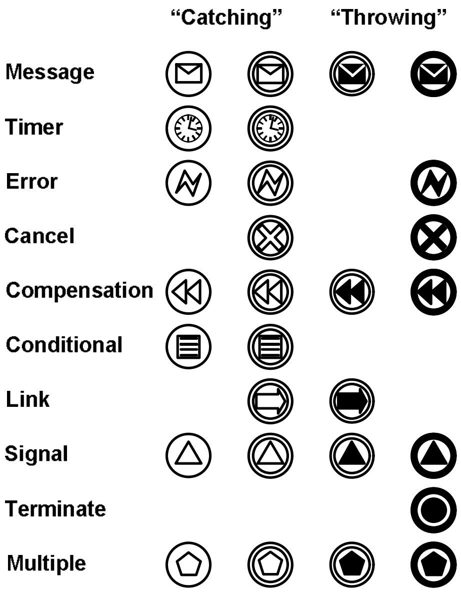
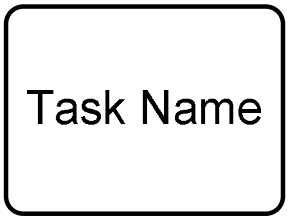
Group (a box around a group of objects within the same category)	A Group is a grouping of Activities that are within the same Category (see page 89). This type of grouping does not affect the Sequence Flow of the Activities within the Group. The Category name appears on the diagram as the group label. Categories can be used for documentation or analysis purposes. Groups are one way in which Categories of objects can be visually displayed on the diagram.	
Text Annotation (attached with an Association)	Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a BPMN Diagram (see page 92).	

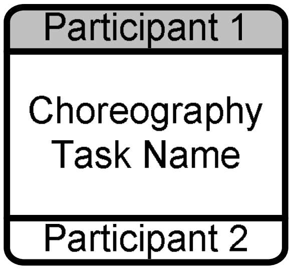
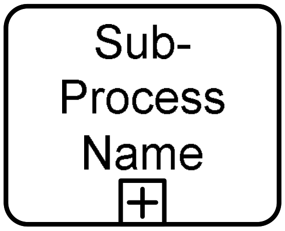
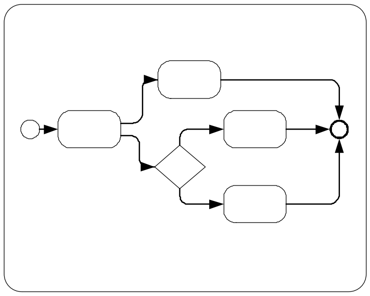
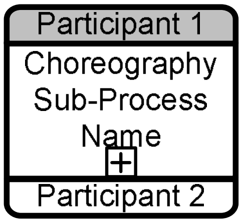
7.2.2. Extended BPMN Modeling Elements

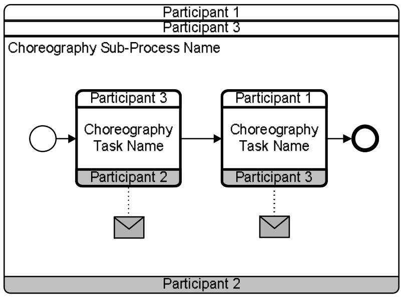
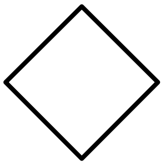
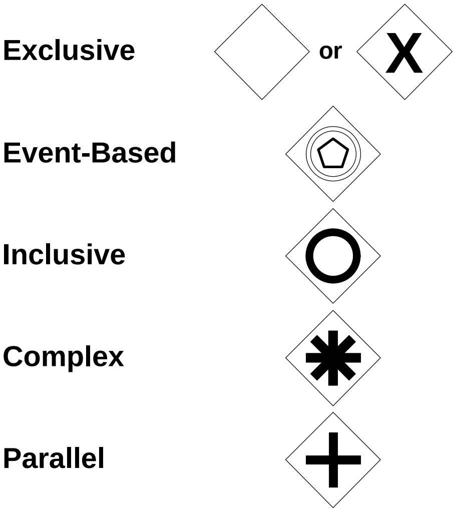
Table 7-2 displays a more extensive list of the Business Process concepts that could be depicted through a business process modeling notation.

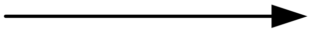
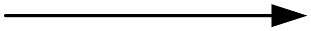
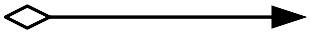
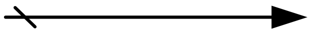
Table 7-2 – BPMN Extended Modeling Elements

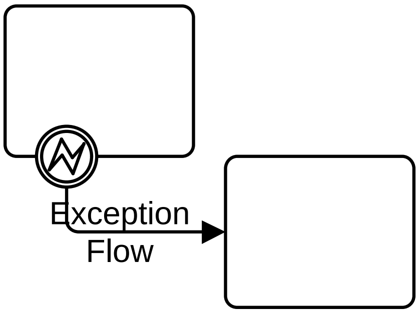
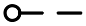
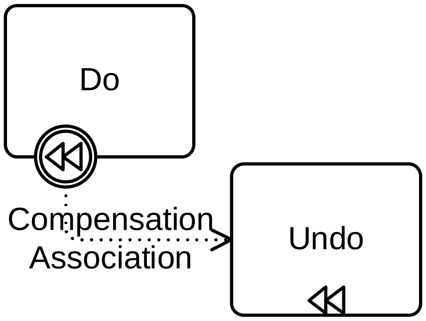
Element	Description	Notation
Event	An Event is something that “happens” during the course of a Process (see page 239) or a Choreography (see page 369). These Events affect the flow of the model and usually have a cause (*Trigger*) or an impact (*Result*). Events are circles with open centers to allow internal markers to differentiate different *Triggers* or *Results*. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End.	
Flow Dimension (e.g., Start, Intermediate, End)		
Start	As the name implies, the Start Event indicates where a particular Process (see page 244) or Choreography (see page 369) will start.	Start 
Intermediate	Intermediate Events occur between a Start Event and an End Event. They will affect the flow of the Process (see page 256) or Choreography (see page 371), but will not start or (directly) terminate the Process.	Intermediate 
End	As the name implies, the End Event indicates where a Process (see page 252) or Choreography (see page 374) will end.	End 

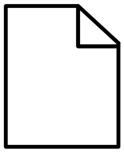
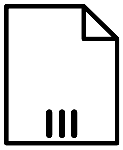
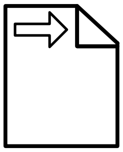
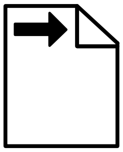
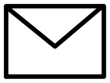
Type Dimension (e.g., None, Message, Timer, Error, Cancel, Compensation, Conditional, Link, Signal, Multiple, Terminate.)	The Start and some Intermediate Events have “triggers” that define the cause for the Event (see “Start Event” on page 244 and “Intermediate Event” on page 256). There are multiple ways that these events can be triggered. End Events may define a “result” that is a consequence of a Sequence Flow ending (see 252). Start Events can only react to (“catch”) a *trigger.* End Events can only create (“throw”) a *result*. Intermediate Events can catch or throw *triggers*. For the Events, *triggers* that catch, the markers are unfilled, and for *triggers* and *results* that throw, the markers are filled. Additionally, some Events, which were used to interrupt Activities in BPMN 1.1, can now be used in a mode that is does not interrupt. The boundary of these Event is dashed (see page 268).	 “Catching” “Throwing” Message Timer Error Cancel Compensation Conditional Link Signal Terminate Multiple
Activity	An Activity is a generic term for work that company performs (see page 159) in Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process Model are: Sub-Process and Task, which are rounded rectangles. Activities are used in both standard Processes and in Choreographies.	
Task (Atomic)	A Task is an atomic Activity that is included within a Process (see page 162). A Task is used when the work in the Process is not broken down to a finer level of Process detail.	

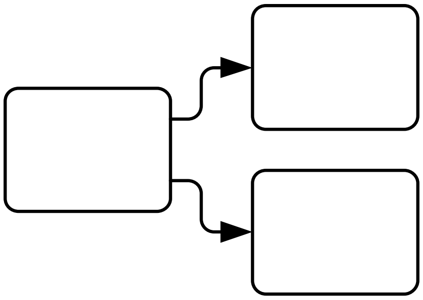
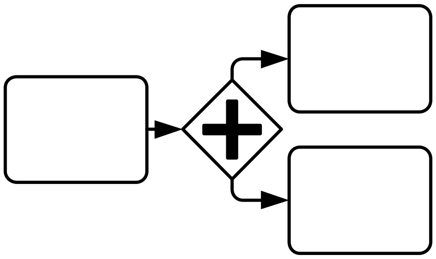
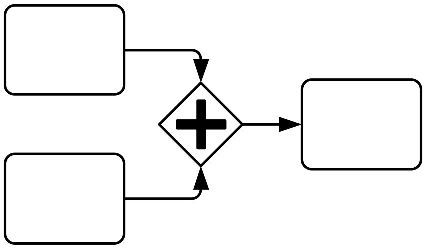
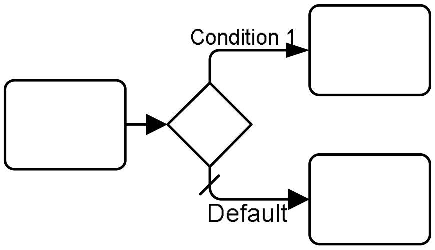
Choreography Task	A Choreography Task is an atomic Activity in a Choreography (see page 350). It represents a set of one (1) or more Message exchanges. Each Choreography Task involves two (2) or more *Participants*. The name of the Choreography Task and each of the *Participants* are all displayed in the different bands that make up the shape's graphical notation. There are two (2) more *Participant* Bands and one Task Name Band.	
Process/Sub-Process (non-atomic)	A Sub-Process is a compound Activity that is included within a Process (see page 176) or Choreography (see page 356). It is compound in that it can be broken down into a finer level of detail (a Process or Choreography) through a set of sub-Activities.	See Next Four (4) Figures
Collapsed Sub-Process	The details of the Sub-Process are not visible in the Diagram (see page 176). A "plus" sign in the lower-center of the shape indicates that the Activity is a Sub-Process and has a lower-level of detail.	
Expanded Sub-Process	The boundary of the Sub-Process is expanded and the details (a Process) are visible within its boundary (see page 176). Note that Sequence Flow cannot cross the boundary of a Sub-Process.	
Collapsed Choreography Sub-Process	The details of the Choreography Sub-Process are not visible in the Diagram (see page 356). A "plus" sign in the lower-center of the Task Name Band of the shape indicates that the Activity is a Sub-Process and has a lower-level of detail.	 Participant 1 Choreography Sub-Process Name Participant 2

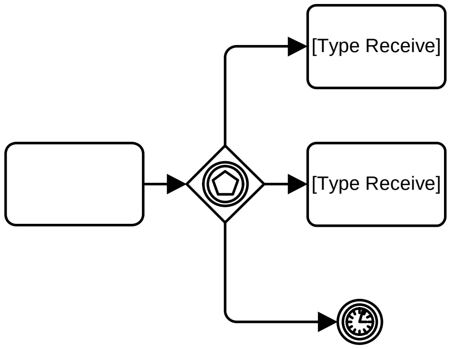
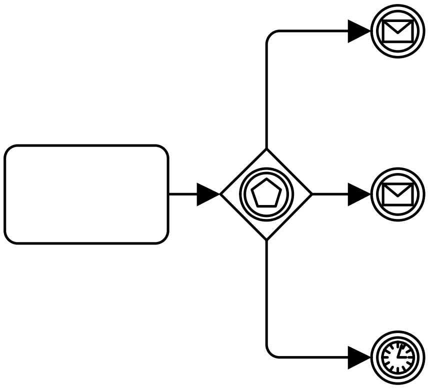
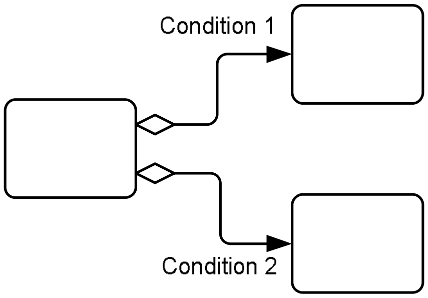
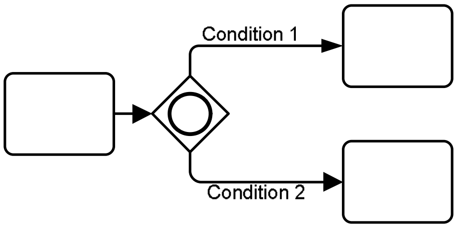
Expanded Choreography Sub-Process	The boundary of the Choreography Sub-Process is expanded and the details (a Choreography) are visible within its boundary (see page 356) Note that Sequence Flow cannot cross the boundary of a Choreography Sub-Process.	 Participant 1 Participant 3 Choreography Sub-Process Name Participant 3 Choreography Task Name Participant 2 Participant 1 Choreography Task Name Participant 3 Participant 2
Gateway	A Gateway is used to control the divergence and convergence of Sequence Flow in a Process (see page 295) and in a Choreography (see page 375). Thus, it will determine branching, forking, merging, and joining of paths. Internal markers will indicate the type of behavior control (see below).	
Gateway Control Types	Icons within the diamond shape of the Gateway will indicate the type of flow control behavior. The types of control include: • Exclusive decision and merging. Both Exclusive (see page 298) and Event-Based (see page 307). Exclusive can be shown with or without the “X” marker. • Inclusive Gateway decision and merging (see page 300) • Complex Gateway -- complex conditions and situations (e.g., 3 out of 5; page 304) • Parallel Gateway forking and joining (see page 302) Each type of control affects both the incoming and outgoing flow.	 Exclusive or X Event-Based Inclusive Complex Parallel
Sequence Flow	A Sequence Flow is used to show the order that Activities will be performed in a Process (see page 159) and in a Choreography (see page 348).	See next seven figures

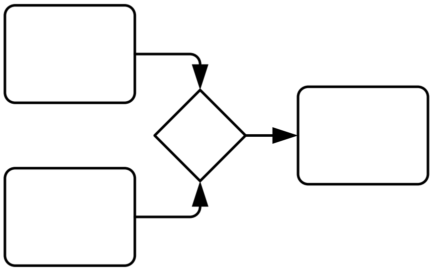
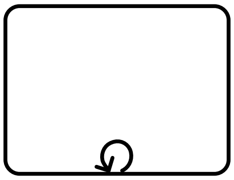
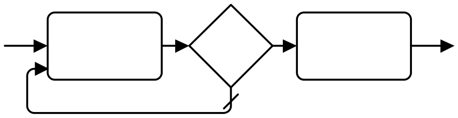
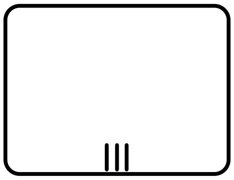
Normal Flow	*Normal flow* refers to the series of Sequence Flow that originates from a Start Event and continues through activities via alternative and parallel paths until it ends at an End Event. This does not include the paths that start from an Intermediate Event attached to the boundary of an Activity.	
Uncontrolled flow	*Uncontrolled flow* refers to flow that is not affected by any conditions or does not pass through a Gateway. The simplest example of this is a single Sequence Flow connecting two Activities. This can also apply to multiple Sequence Flow that converge to or diverge from an Activity. For each uncontrolled Sequence Flow a *token* will flow from the source object through the Sequence Flow to the target object.	
Conditional flow	Sequence Flow can have condition Expressions that are evaluated at runtime to determine whether or not the Sequence Flow will be used (i.e., will a *token* travel down the Sequence Flow – see page 109). If the *conditional flow* is outgoing from an Activity, then the Sequence Flow will have a mini-diamond at the beginning of the connector (see figure to the right). If the *conditional flow* is outgoing from a Gateway, then the line will not have a mini-diamond (see figure in the row above).	
Default flow	For Data-Based Exclusive Gateways or Inclusive Gateways, one type of flow is the Default *condition flow* (see page 109). This flow will be used only if all the other outgoing *conditional flow* is not *true* at runtime. These Sequence Flow will have a diagonal slash will be added to the beginning of the connector (see the figure to the right).	

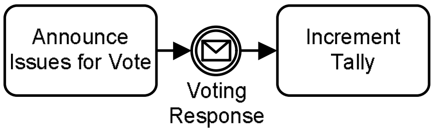
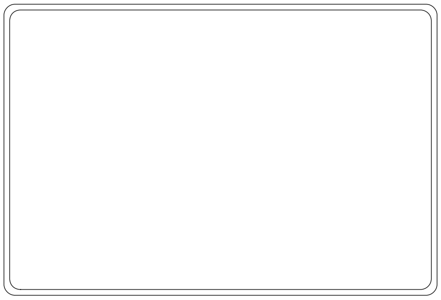
Exception Flow	*Exception flow* occurs outside the *normal flow* of the Process and is based upon an Intermediate Event attached to the boundary of an Activity that occurs during the performance of the Process (see page 285).	
Message Flow	A Message Flow is used to show the flow of Messages between two *Participants* that are prepared to send and receive them (see page 119). In BPMN, two separate Pools in a Collaboration Diagram will represent the two *Participants* (e.g., PartnerEntities and/or PartnerRoles).	
Compensation Association	*Compensation* Association occurs outside the *normal flow* of the Process and is based upon a Compensation Intermediate Event that is triggered through the failure of a *transaction* or a *throw* Compensate Event (see page 270). The target of the Association must be marked as a Compensation Activity.	

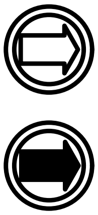
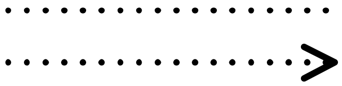
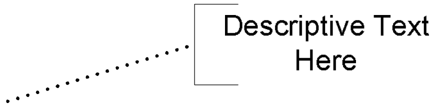
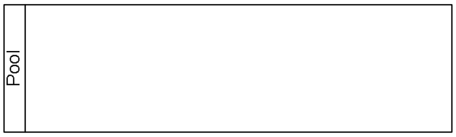
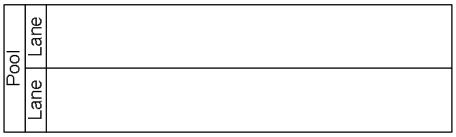
Data Object	Data Objects provide information about what Activities require to be performed and/or what they produce (see page 213), Data Objects can represent a singular object or a collection of objects. Data Input and Data Output provide the same information for Processes.	 ***Collection***  Data Input Data Output  
Message	A Message is used to depict the contents of a communication between two *Participants* (as defined by a business PartnerRole or a business PartnerEntity—see on page 112).	

Fork	BPMN uses the term “fork” to refer to the dividing of a path into two or more parallel paths (also known as an AND-Split). It is a place in the Process where activities can be performed concurrently, rather than sequentially. There are two options: Multiple Outgoing Sequence Flow can be used (see figure top-right). This represents “uncontrolled” flow is the preferred method for most situations. A Parallel Gateway can be used (see figure bottom-right). This will be used rarely, usually in combination with other Gateways.	 
Join	BPMN uses the term “join” to refer to the combining of two or more parallel paths into one path (also known as an AND-Join or synchronization). A Parallel Gateway is used to show the joining of multiple Sequence Flow.	
Decision, Branching Point	Decisions are Gateways within a Process (see page 295) or a Choreography (see page 375) where the flow of control can take one or more alternative paths.	See next five rows.
Exclusive	This Decision represents a branching point where Alternatives are based on conditional expressions contained within the *outgoing* Sequence Flow (see page 298 or page 375). Only one of the Alternatives will be chosen.	

Event-Based	This Decision represents a branching point where Alternatives are based on an Event that occurs at that point in the Process (see page 307) or Choreography (see page 375). The specific Event, usually the receipt of a Message, determines which of the paths will be taken. Other types of Events can be used, such as Timer. Only one of the Alternatives will be chosen. There are two options for receiving Messages: Tasks of Type Receive can be used (see figure top-right). Intermediate Events of Type Message can be used (see figure bottom-right).	 
Inclusive	This Decision represents a branching point where Alternatives are based on conditional expressions contained within the outgoing Sequence Flow (see page 300). In some sense it is a grouping of related independent Binary (Yes/No) Decisions. Since each path is independent, all combinations of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken. A Default Condition could be used to ensure that at least one path is taken. There are two versions of this type of Decision: The first uses a collection of conditional Sequence Flow, marked with mini-diamonds (see top-right figure). The second uses an Inclusive Gateway (see bottom-right picture).	 

Merging	BPMN uses the term “merge” to refer to the exclusive combining of two or more paths into one path (also known as an OR-Join). A Merging Exclusive Gateway is used to show the merging of multiple Sequence Flow. If all the incoming flow is alternative, then a Gateway is not needed. That is, uncontrolled flow provides the same behavior.	
Looping	BPMN provides 2 (two) mechanisms for looping within a Process.	See Next Two Figures
Activity Looping	The attributes of Tasks and Sub-Processes will determine if they are repeated or performed once (see page 198). There are two types of loops: Standard and Multi-Instance. A small looping indicator will be displayed at the bottom-center of the activity.	
Sequence Flow Looping	Loops can be created by connecting a Sequence Flow to an “upstream” object. An object is considered to be upstream if that object has an outgoing Sequence Flow that leads to a series of other Sequence Flow, the last of which is an incoming Sequence Flow for the original object.	
Multiple Instances	The attributes of Tasks and Sub-Processes will determine if they are repeated or performed once (see page 198). A small parallel indicator will be displayed at the bottom-center of the activity.	

Process Break (something out of the control of the process makes the process pause)	A Process Break is a location in the Process that shows where an expected delay will occur within a Process (see page 256). An Intermediate Event is used to show the actual behavior (see top-right figure). In addition, a Process Break Artifact, as designed by a modeler or modeling tool, can be associated with the Event to highlight the location of the delay within the flow.	 Announce Issues for Vote Voting Response Increment Tally
Transaction	A transaction is a Sub-Process that is supported by a special protocol that insures that all parties involved have complete agreement that the activity should be completed or cancelled (see page 188). The attributes of the activity will determine if the activity is a transaction. A double-lined boundary indicates that the Sub-Process is a Transaction.	
Nested/Embedded Sub-Process (Inline Block)	A nested (or embedded) Sub-Process is an activity that shares the same set of data as its parent process (see page 176). This is opposed to a Sub-Process that is independent, re-usable, and referenced from the parent process. Data needs to be passed to the referenced Sub-Process, but not to the nested Sub-Process.	There is no special indicator for nested Sub-Processes
Group (a box around a group of objects within the same category)	A Group is a grouping of Activities that are within the same *Category* (see page 89). This type of grouping does not affect the Sequence Flow of the Activities within the Group. The *Category* name appears on the diagram as the group label. *Categories* can be used for documentation or analysis purposes. Groups are one way in which *Categories* of objects can be visually displayed on the diagram.	

Off-Page Connector	Generally used for printing, this object will show where the Sequence Flow leaves one page and then restarts on the next page. A Link Intermediate Event can be used as an Off-Page Connector.	
Association	An Association is used to link information and Artifacts with BPMN graphical elements (see page 88). Text Annotations (see page 93) and other Artifacts (see page 86) can be Associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.	
Text Annotation (attached with an Association)	Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a BPMN Diagram (see page 92).	
Pool	A Pool is the graphical representation of a *Participant* in a Collaboration (see page 146). It is also acts as a “swimlane” and a graphical container for partitioning a set of Activities from other Pools, usually in the context of B2B situations.	
Lanes	A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally (see on page 149). Lanes are used to organize and categorize Activities.	

7.3. BPMN Diagram Types

The BPMN 2.0 aims to cover three basic models of Processes: *private* Processes (both *executable* and *non-executable*), *public* Processes, and Choreographies. Within and between these three BPMN sub-models, many types of Diagrams can be created. The following are examples of Business Processes that can be modeled using BPMN 2.0:

- High-level *non-executable* Process Activities (not functional breakdown)
- Detailed executable Business Process
- As-is or old Business Process
- To-be or new Business Process
- A description of expected behavior between two (2) or more business *Participants*—a Choreography.
- Detailed *private* Business Process (either *executable* or *non-executable*) with interactions to one or more external *Entities* (or "Black Box" Processes)
- Two or more detailed *executable* Processes interacting
- Detailed *executable* Business Process relationship to a Choreography
- Two or more *public* Processes
- *Public* Process relationship to Choreography
- Two or more detailed *executable* Business Processes interacting through a Choreography

BPMN is designed to allow describing all above examples of Business Processes. However, the ways that different sub-models are combined is left to tool vendors. A BPMN 2.0 compliant implementation may recommend that modelers pick a focused purpose, such as a *private* Process, or Choreographies. However, the BPMN 2.0 specification makes no assumptions.

7.4. Use of Text, Color, Size, and Lines in a Diagram

Text Annotation objects can be used by the modeler to display additional information about a Process or attributes of the objects within a BPMN Diagram.

- ◆ Flow objects and Flow MAY have labels (e.g., its name and/or other attributes) placed inside the shape, or above or below the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor.
- ◆ The fills that are used for the graphical elements MAY be white or clear.
  - ◆ The notation MAY be extended to use other fill colors to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute). However,
    - ◆ The markers for "throwing" Events MUST have a dark fill (see "End Event" on page 252 and "Intermediate Event" on page 256 for more details).
    - ◆ Participant Bands for Choreography Tasks and Choreography Sub-Processes that are *not* the initiator of the Activity MUST have a light fill (see "Choreography Task"

on page 350 and “Choreography Sub-Process” on page 356 for more details).

- Flow objects and markers MAY be of any size that suits the purposes of the modeler or modeling tool.
- The lines that are used to draw the graphical elements MAY be black.
  - The notation MAY be extended to use other line colors to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute).
  - The notation MAY be extended to use other line styles to suit the purpose of the modeler or tool (e.g., to highlight the value of an object attribute) with the condition that the line style MUST NOT conflict with any current BPMN defined line style. Thus, the line styles of Sequence Flow, Message Flow, and Text Associations MUST NOT be modified or duplicated.

7.5. Flow Object Connection Rules

An *incoming* Sequence Flow can connect to any location on a Flow Object (left, right, top, or bottom). Likewise, an *outgoing* Sequence Flow can connect from any location on a Flow Object (left, right, top, or bottom). A Message Flow also has this capability. BPMN allows this flexibility; however, we also recommend that modelers use judgment or best practices in how Flow Objects should be connected so that readers of the Diagrams will find the behavior clear and easy to follow. This is even more important when a Diagram contains Sequence Flow and Message Flow. In these situations it is best to pick a direction of Sequence Flow, either left to right or top to bottom, and then direct the Message Flow at a 90° angle to the Sequence Flow. The resulting Diagrams will be much easier to understand.

7.5.1. Sequence Flow Connections Rules

Table 8.4 displays the BPMN Flow Objects and shows how these objects can connect to one another through Sequence Flow. These rules apply to the connections within a Process Diagram and within a Choreography Diagram. The ↗ symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into and out of an object is subject to various configuration dependencies are not specified here. Refer to the sections in the next chapter for each individual object for more detailed information on the appropriate connection rules. *Note that if a Sub-Process has been expanded within a Diagram, the objects within the Sub-Process cannot be connected to objects outside of the Sub-Process. Nor can Sequence Flow cross a Pool boundary.*

Table 7-3 – Sequence Flow Connection Rules

From\To	○	▢	⊞	◇	◎	**◯**
○		↗	↗	↗	↗	↗
▢		↗	↗	↗	↗	↗
⊞		↗	↗	↗	↗	↗
◇		↗	↗	↗	↗	↗
◎		↗	↗	↗	↗	↗
◯						

Note – Only those objects that can have *incoming* and/or *outgoing* Sequence Flow are shown in the table. Thus, Pool, Lane, Data Object, Group, and Text Annotation are not listed in the table. Also, the Activity shapes in the table represent Activities and Sub-Processes for Processes, and Choreography Activities and Choreography Sub-Processes for Choreography.

7.5.2. Message Flow Connection Rules

Table 8.5 displays the BPMN modeling objects and shows how these objects can connect to one another through Message Flow. These rules apply to the connections within a Collaboration Diagram. The ⇗ symbol indicates that the object listed in the row can connect to the object listed in the column. The quantity of connections into and out of an object is subject to various configuration dependencies are not specified here. Refer to the sections in the next chapter for each individual object for more detailed information on the appropriate connection rules. *Note that Message Flow cannot connect to objects that are within the same Pool.*

Table 7-4 – Message Flow Connection Rules

From\To	✉	Name Pool	▢	⊞	◎	●
✉						
Name Pool	⇗	⇗	⇗	⇗	⇗	
▢	⇗	⇗	⇗	⇗	⇗	
⊞	⇗	⇗	⇗	⇗	⇗	
◎	⇗	⇗	⇗	⇗	⇗	
●	⇗	⇗	⇗	⇗	⇗	

Note – Only those objects that can have *incoming* and/or *outgoing* Message Flow are shown in the table. Thus, Lane, Gateway, Data Object, Group, and Text Annotation are not listed in the table.

7.6. BPMN Extensibility

BPMN 2.0 introduces an extensibility mechanism that allows extending standard BPMN elements with additional attributes. It can be used by modelers and modeling tools to add non-standard elements or Artifacts to satisfy a specific need, such as the unique requirements of a vertical domain, and still have valid BPMN Core. Extension attributes must not contradict the semantics of any BPMN element. In addition, while extensible, BPMN Diagrams should still have the basic look-and-feel so that a Diagram by any modeler should be easily understood by any viewer of the Diagram. Thus the footprint of the basic flow elements (Events, Activities, and Gateways) must not be altered.

The specification differentiates between mandatory and optional extensions (Section 8.2.3 explains the syntax used to declare extensions). If a mandatory extension is used, a compliant implementation must understand the extension. If an optional extension is used, a compliant implementation may ignore the extension.

7.7. BPMN Example

The following is an example of a manufacturing process from different perspectives.

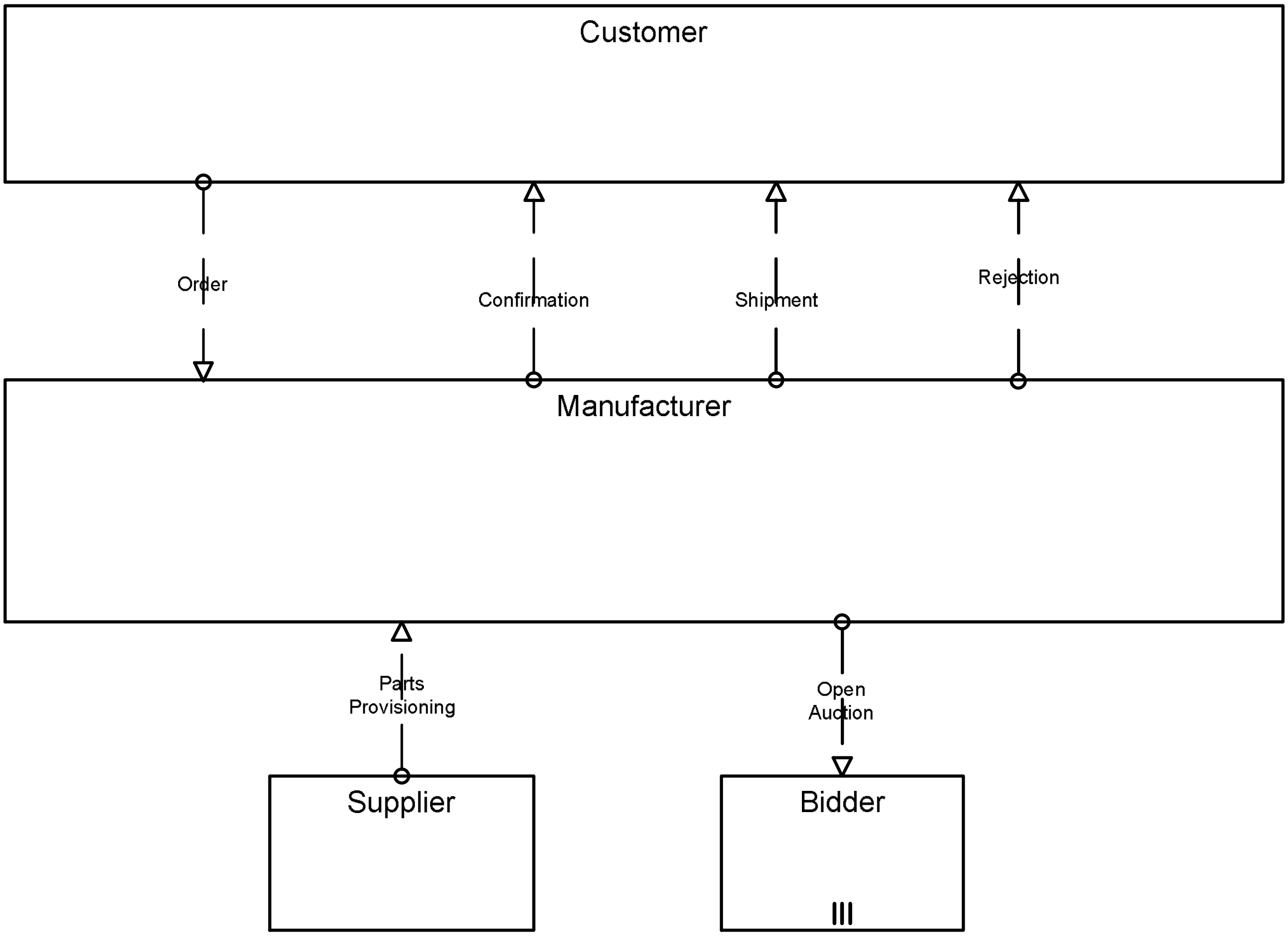


Figure 7-6 – An example of a Collaboration diagram with black-box Pools

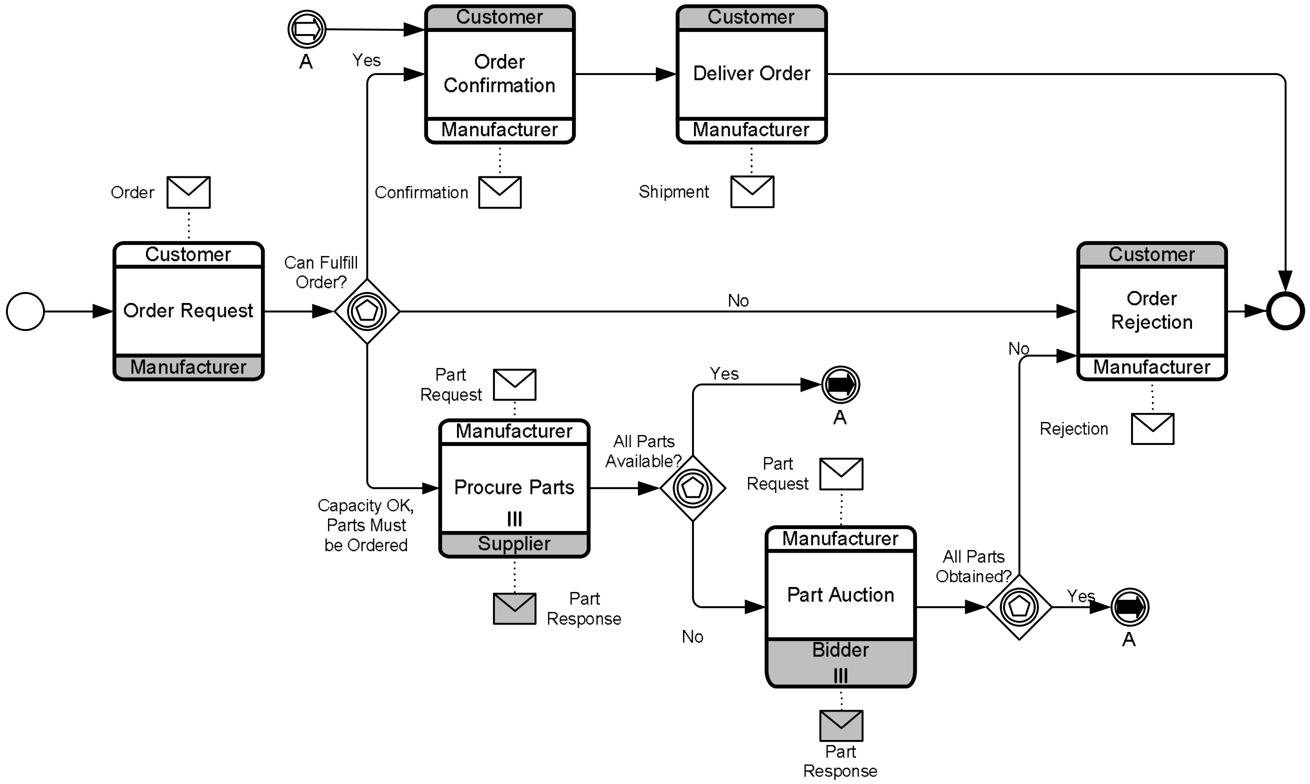


Figure 7-7 – An example of a stand-alone Choreography diagram

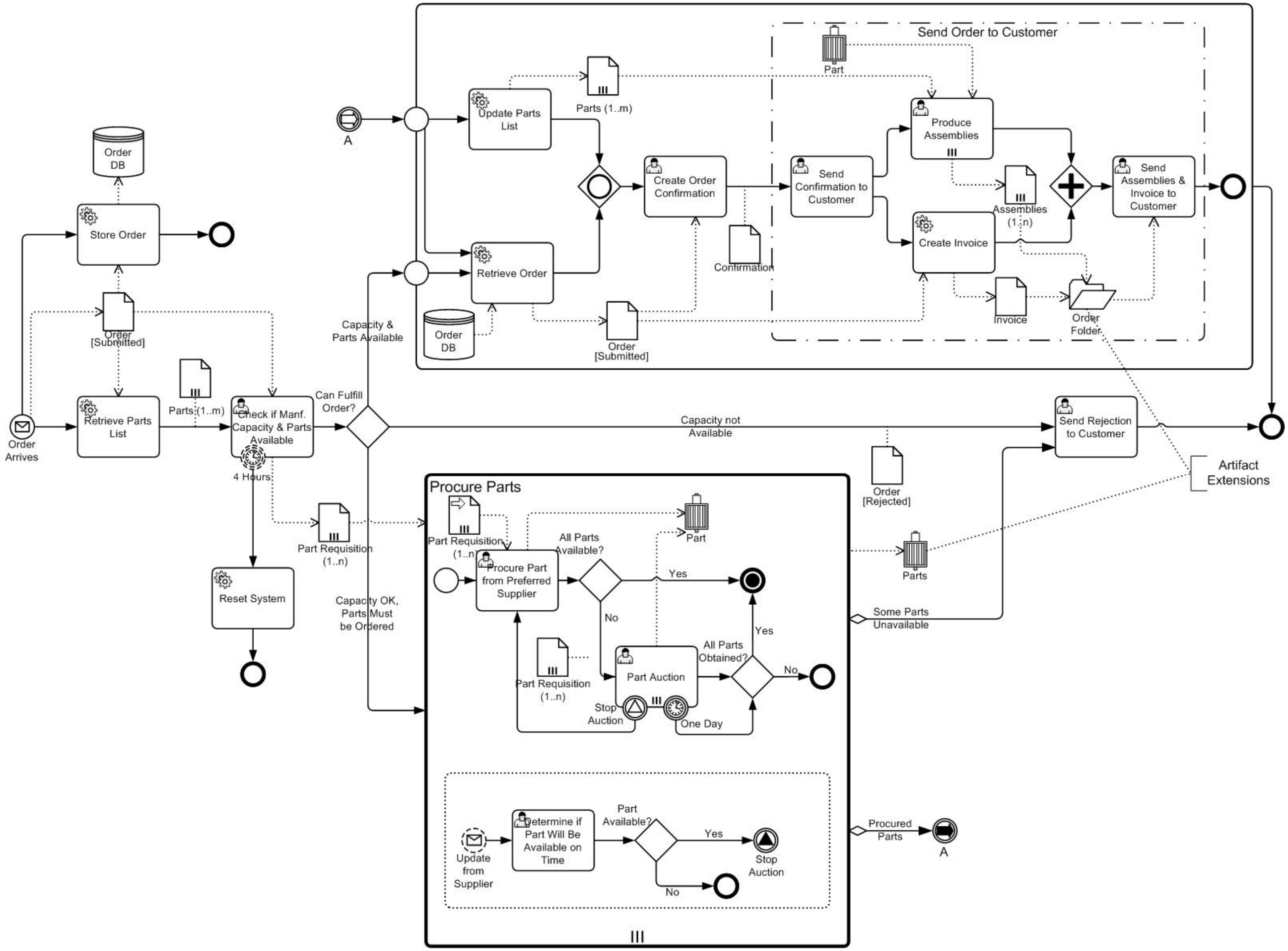


Figure 7-8 – An example of a stand-alone Process (Orchestration) diagram

8. BPMN Core Structure

Note: The content of this chapter is required for all BPMN conformance types. For more information about BPMN conformance types, see page 28.

The proposed technical structuring of BPMN is based on the concept of extensibility layers on top of a basic series of simple elements identified as *Core Elements* of the specification. From this core set of constructs, layering is used to describe additional elements of the specification that extend and add new constructs to the specification and relies on clear dependency paths for resolution. The XML Schema model lends itself particularly well to the proposed structuring model with formalized import and resolution mechanics that remove ambiguities in the definitions of elements in the outer layers of the specification.

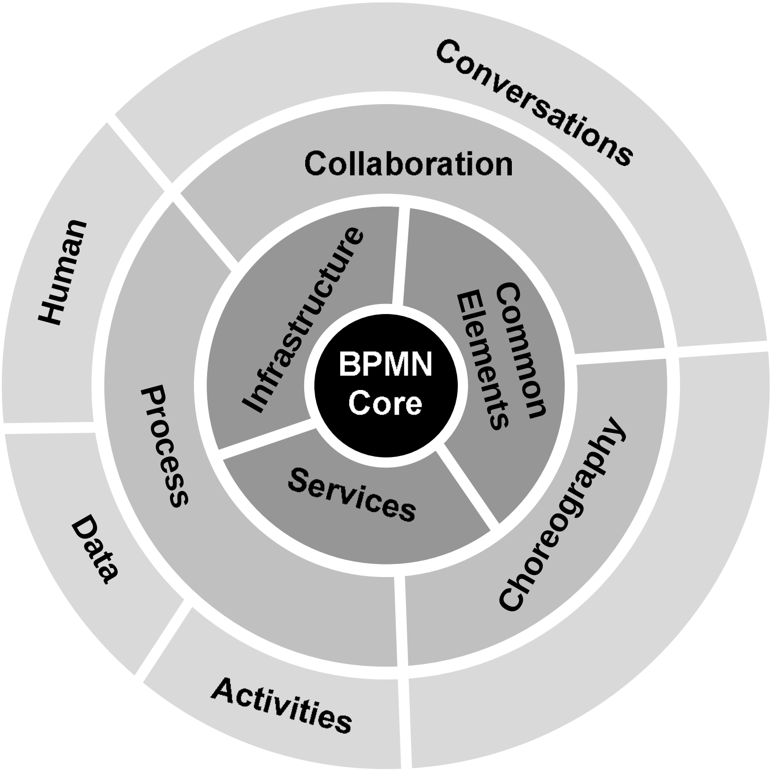


Figure 8-1 – A representation of the BPMN Core and Layer Structure

Figure 8-1 shows the basic principles of layering that can be composed in well defined ways. The approach uses formalization constructs for extensibility that are applied consistently to the definition.

The additional effect of layering is that compatibility layers can be built, allowing for different levels of compliance amongst vendors, and also enabling vendors to add their own layers in support of different vertical industries or target audiences. In addition, it provides mechanism for the redefinition of previously existing concepts without affecting backwards compatibility, but defining two or more non-composable layers, the level of compliance with the specification and backwards compatibility can be achieved without compromising clarity.

The BPMN specification is structured in layers, where each layer builds on top of and extends lower layers. Included is a *Core* or kernel which includes the most fundamental elements of BPMN that are required for constructing BPMN diagrams: Process, Choreography, Collaboration, and Conversation. The *Core* is intended to be simple, concise, and extendable, with well defined behavior

The *Core* contains three (3) sub-packages (see Figure 8-2):

- Foundation: The fundamental constructs needed for BPMN semantic modeling.
- Service: The fundamental constructs needed for modeling services and interfaces.
- Common: Those classes which are common to the layers of Process, Choreography, and Collaboration.

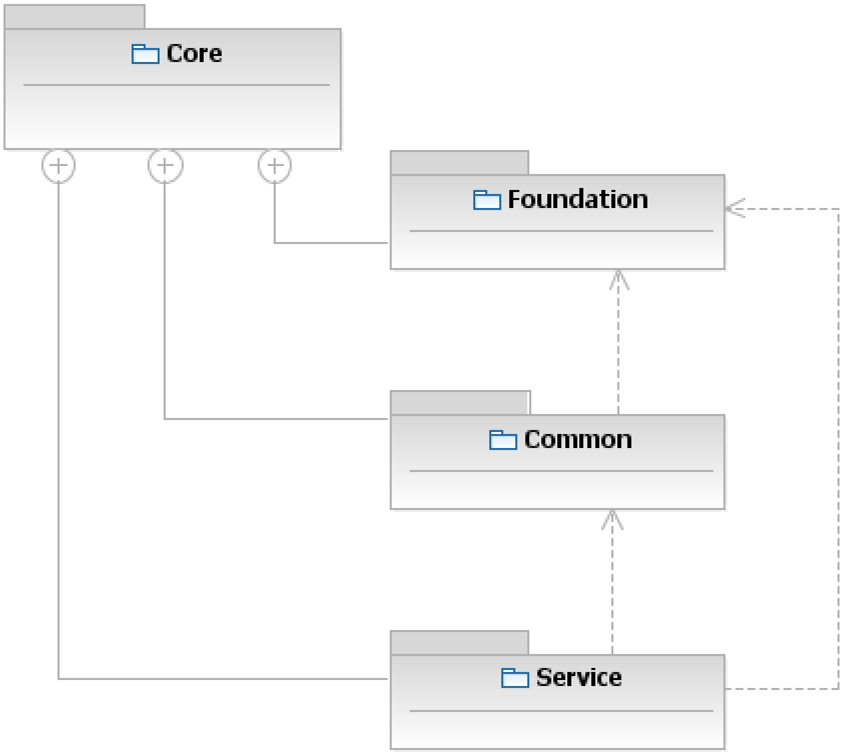


Figure 8-2 – Class diagram showing the core packages

Figure 8-3 displays the organization of the main set of BPMN core model elements.

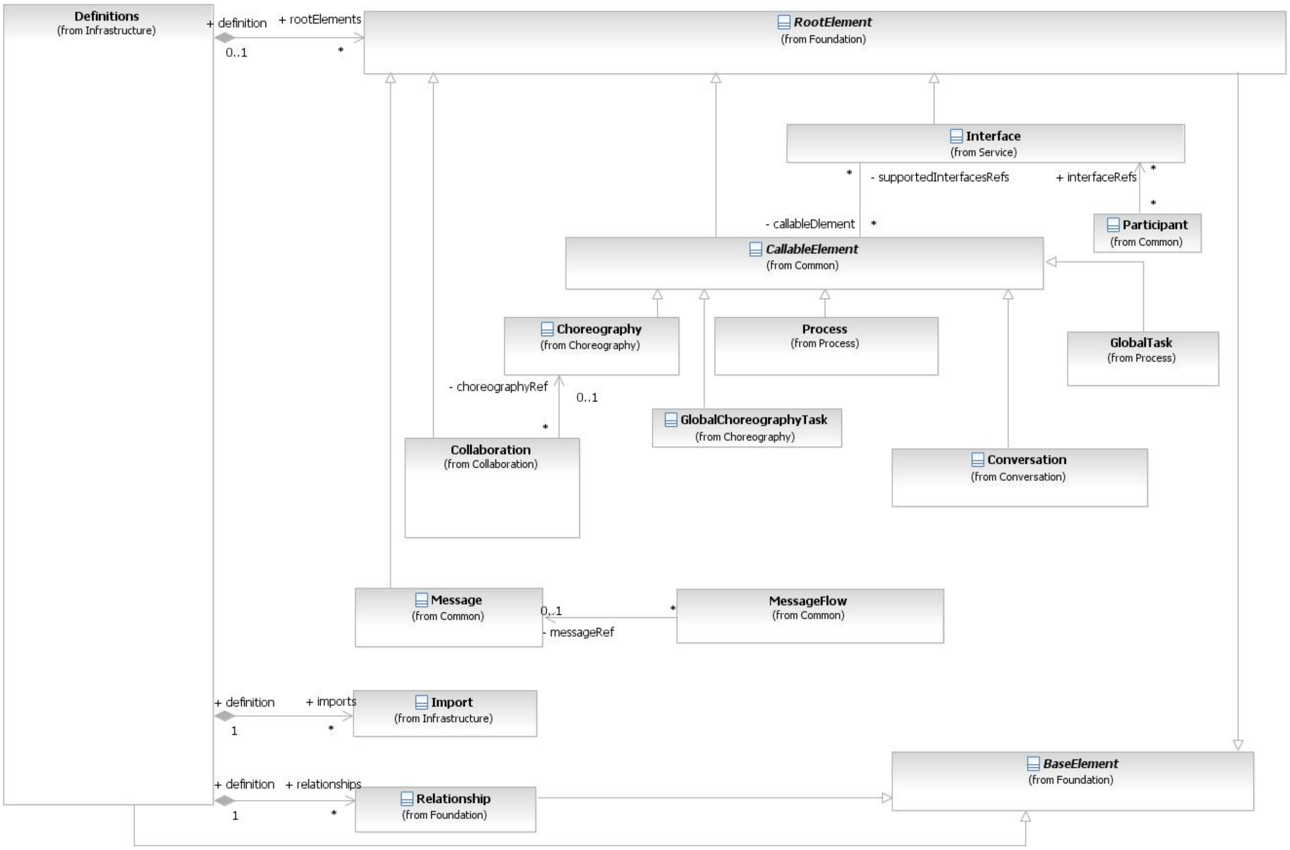


Figure 8-3 – Class diagram showing the organization of the core BPMN elements

8.1. Infrastructure

The BPMN `Infrastructure` package contains two (2) elements that are used for both semantic models and diagram models.

8.1.1. Definitions

The `Definitions` class is the outermost containing object for all BPMN elements. It defines the scope of visibility and the namespace for all contained elements. The interchange of BPMN files will always be through one or more `Definitions`.

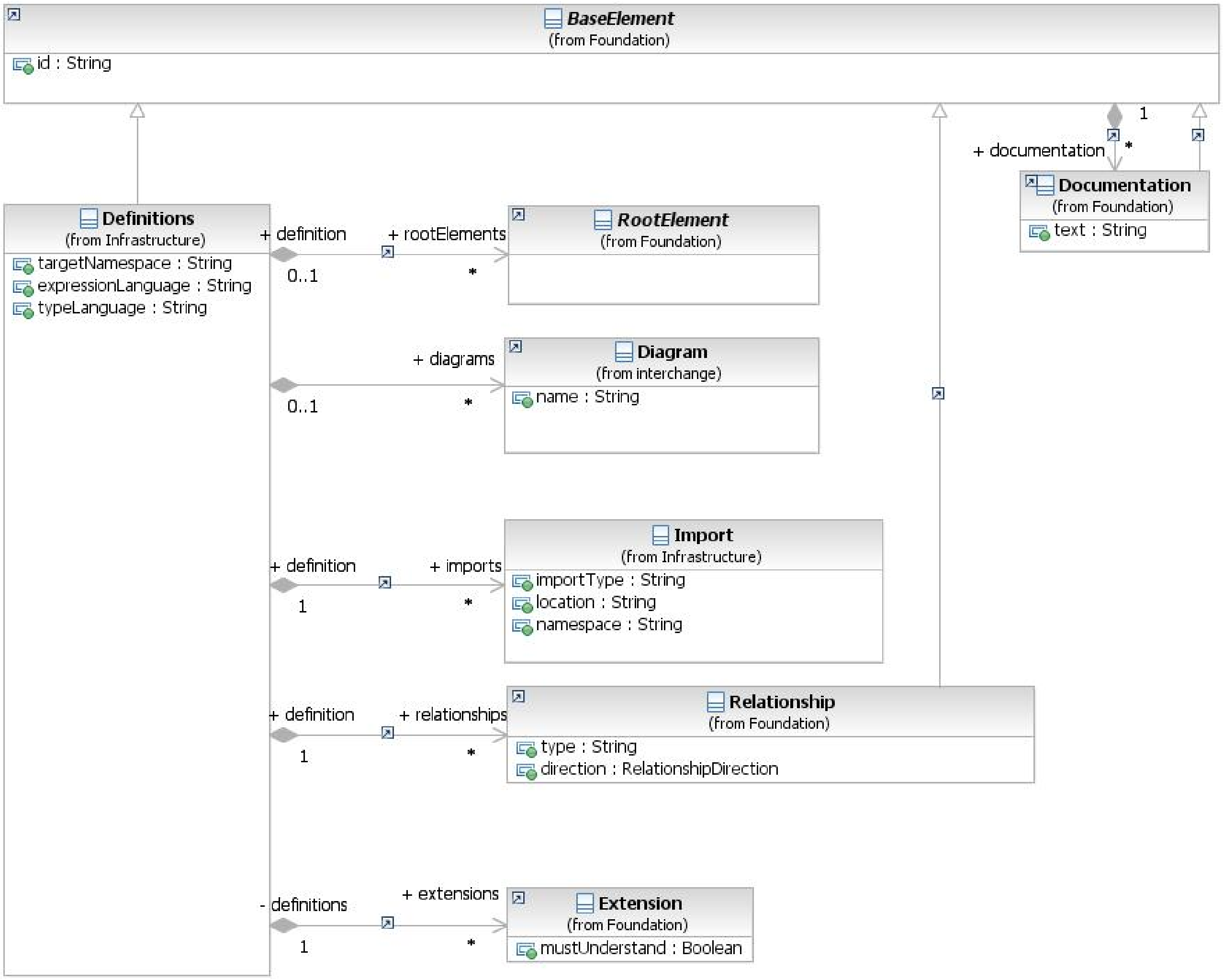


Figure 8-4 – Definitions class diagram

The `Definitions` element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 8-1 presents the additional attributes and model associations of the `Definitions` element:

Table 8-1 – Definitions attributes and model associations

Attribute Name	Description/Usage
targetNamespace: string	This attribute identifies the namespace associated with the `Definition` and follows the convention established by XML Schema.
expressionLanguage: string [0..1]	This attribute identifies the formal expression language used in `Expressions` within the elements of this `Definition`. The Default is “http://www.w3.org/1999/XPath”. This value may be overridden on each individual formal expression.

typeLanguage: string [0..1]	This attribute identifies the type system used by the elements of this Definition. Defaults to http://www.w3.org/2001/XMLSchema. This value can be overridden on each individual ItemDefinition.
rootElements: RootElement [0..*]	This attribute lists the root elements that are at the root of this Definitions. These elements can be referenced within this Definitions and are visible to other Definitions.
diagrams: Diagram [0..*]	This attribute lists the Diagrams that are contained within this Definitions (see page 398 for more information on Diagrams).
imports: Import [0..*]	This attribute is used to import externally defined elements and make them available for use by elements within this Definitions.
extensions: Extension [0..*]	This attribute identifies extensions beyond the attributes and model associations in the base BPMN specification. See page 77 for additional information on extensibility
relationships: Relationship [0..*]	This attribute enables the extension and integration of BPMN models into larger system/development Processes.

8.1.2. Import

The Import class is used when referencing external element, either BPMN elements contained in other BPMN Definitions or non-BPMN elements. Imports must be explicitly defined.

Table 8-2 presents the attributes of Import:

Table 8-2 – Import attributes

Attribute Name	Description/Usage
importType: string	Specifies the style of import associated with the element. For example, a value of “http://www.w3.org/2001/XMLSchema” indicates that the imported element is an XML schema. A value of http://www.omg.org/bpmn20 indicates that the imported element is a BPMN element.
location: string [0..1]	Identifies the location of the imported element.
namespace: string	Identifies the namespace of the imported element.

8.1.3. Infrastructure Package XML Schemas

Table 8-3 – Definitions XML schema

```
<xsd:element name="definitions" type="tDefinitions"/>
<xsd:complexType name="tDefinitions">
    <xsd:sequence>
        <xsd:element ref="import" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="extension" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="rootElement" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="di:diagram" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="relationship" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI" use="optional"
            default="http://www.w3.org/1999/XPath"/>
    <xsd:attribute name="typeLanguage" type="xsd:anyURI" use="optional"
            default="http://www.w3.org/2001/XMLSchema"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
```

Table 8-4 – Import XML schema

```
<xsd:element name="import" type="tImport"/>
<xsd:complexType name="tImport">
    <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="location" type="xsd:string" use="required"/>
    <xsd:attribute name="importType" type="xsd:anyURI" use="required"/>
</xsd:complexType>
```

8.2. Foundation

The `Foundation` package contains classes which are shared amongst other packages in the `Core` (see Figure 8-5) of a semantic model.

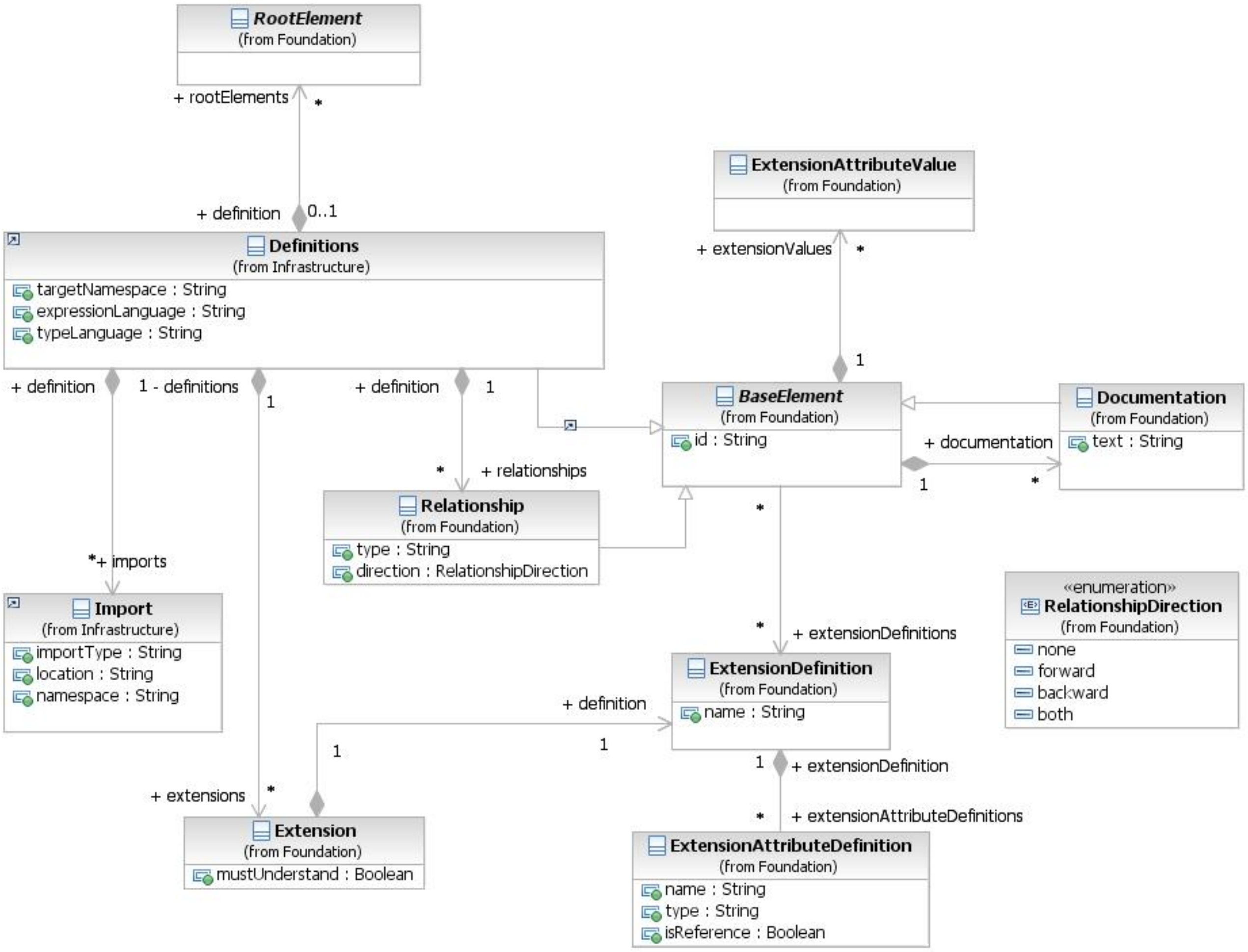


Figure 8-5 – Classes in the Infrastructure package

8.2.1. Base Element

`BaseElement` is the abstract super class for most BPMN elements. It provides the attributes `id` and `documentation`, which other elements will inherit.

Table 8-5 presents the attributes and model associations for the BaseElement.

Table 8-5 – BaseElement attributes and model associations

Attribute Name	Description/Usage
id: string	This attribute is used to uniquely identify BPMN elements.
documentation: Documentation [0..*]	This attribute is used to annotate the BPMN element, such as descriptions and other documentation.
extensionDefinitions: ExtensionDefinition [0..*]	This attribute is used to attach additional attributes and associations to any BaseElement. This association is not applicable when the XML schema interchange is used, since the XSD mechanisms for supporting anyAttribute and any element already satisfy this requirement. See page 77 for additional information on extensibility.
extensionValues: ExtensionAttributeValue [0..*]	This attribute is used to provide values for extended attributes and model associations. This association is not applicable when the XML schema interchange is used, since the XSD mechanisms for supporting anyAttribute and any element already satisfy this requirement. See page 77 for additional information on extensibility.

8.2.2. Documentation

All BPMN elements that inherit from the BaseElement will have the capability, through the Documentation element, to have one (1) or more text descriptions of that element.

The Documentation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-6 presents the additional attributes of the Documentation element:

Table 8-6 – Documentation attributes

Attribute Name	Description/Usage
text: string	This attribute is used to capture the text descriptions of a BPMN element.

8.2.3. Extensibility

The BPMN metamodel is aimed to be extensible. This allows BPMN adopters to extend the specified metamodel in a way that allows them to be still BPMN-compliant.

It provides a set of extension elements, which allows BPMN adopters to attach additional attributes and elements to standard and existing BPMN elements.

This approach results in more interchangeable models, because the standard elements are still intact and can still be understood by other BPMN adopters. It's only the additional attributes and elements that may be lost during interchange.

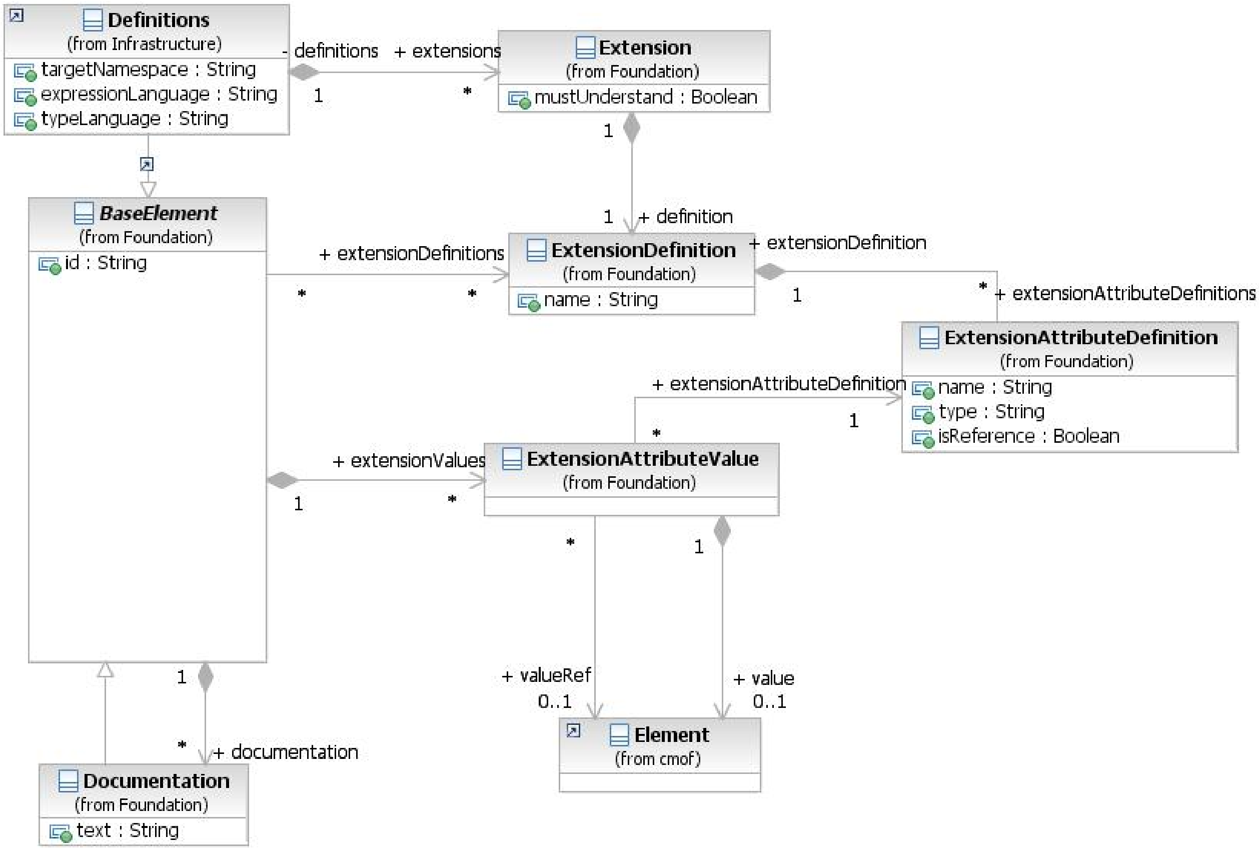


Figure 8-6 – Extension class diagram

A BPMN Extension basically consists of four different elements:

- Extension
- ExtensionDefinition
- ExtensionAttributeDefinition
- ExtensionAttributeValue

The core elements of an Extension are the `ExtensionDefinition` and `ExtensionAttributeDefinition`. The latter defines a list of attributes which can be attached to any BPMN element. The attribute list defines the name and type of the new attribute. This allows BPMN adopters to integrate any meta model into the BPMN meta model and reuse already existing model elements.

The `ExtensionDefinition` itself can be created independent of any BPMN element or any BPMN definition.

In order to use an `ExtensionDefinition` within a BPMN model definition (`Definitions` element), the `ExtensionDefinition` must be associated with an `Extension` element which binds the

ExtensionDefinition to a specific BPMN model definition. The Extension element itself is contained within the BPMN element Definitions and therefore available to be associated with any BPMN element making use of the ExtensionDefinition.

Every BPMN element which subclasses the BPMN BaseElement can be extended by additional attributes. This works by associating a BPMN element with an ExtensionDefinition which was defined at the BPMN model definitions level (element Definitions).

Additionally, every "extended" BPMN element contains the actual extension attribute value. The attribute value, defined by the element ExtensionAttributeValue contains the value of type Element. It also has an association to the corresponding attribute definition.

Extension

The Extension element binds/imports an ExtensionDefinition and its attributes to a BPMN model definition.

Table 8-7 presents the attributes and model associations for the Extension element:

Table 8-7 – Extension attributes and model associations

Attribute Name	Description/Usage
mustUnderstand: boolean [0..1] = False	This flag defines if the semantics defined by the extension definition and its attribute definition must be understood by the BPMN adopter in order to process the BPMN model correctly. Defaults to False.
definition: ExtensionDefinition	Defines the content of the extension. Note that in the XML schema, this definition is provided by an external XML schema file and is simply referenced by QName.

ExtensionDefinition

The ExtensionDefinition class defines and groups additional attributes. This type is not applicable when the XML schema interchange is used, since XSD Complex Types already satisfy this requirement.

Table 8-8 presents the attributes and model associations for the ExtensionDefinition element:

Table 8-8 – ExtensionDefinition attributes and model associations

Attribute Name	Description/Usage
name: string	The name of the extension. This is used as a namespace to uniquely identify the extension content.
extensionAttributeDefinitions: ExtensionDefinition [0..*]	The specific attributes that make up the extension.

ExtensionAttributeDefinition

The `ExtensionAttributeDefinition` defines new attributes. This type is not applicable when the XML schema interchange is used; since the XSD mechanisms for supporting “AnyAttribute” and “Any” type already satisfy this requirement.

Table 8-9 presents the attributes for the `ExtensionAttributeDefinition` element:

Table 8-9 – ExtensionAttributeDefinition attributes

Attribute Name	Description/Usage
name: string	The `name` of the extension attribute.
type: string	The type that is associated with the attribute.
isReference: boolean [0..1] = False	Indicates if the attribute value will be referenced or contained.

ExtensionAttributeValue

The `ExtensionAttributeValue` contains the attribute value. This type is not applicable when the XML schema interchange is used; since the XSD mechanisms for supporting “AnyAttribute” and “Any” type already satisfy this requirement.

Table 8-10 presents the model associations for the `ExtensionAttributeValue` element:

Table 8-10 – ExtensionAttributeValue model associations

Attribute Name	Description/Usage
value: Element [0..1]	The contained attribute value, used when the associated `ExtensionAttributeDefinition.isReference` is false. The type of this `Element` must conform to the type specified in the associated `ExtensionAttributeDefinition`.
valueRef: Element [0..1]	The referenced attribute value, used when the associated `ExtensionAttributeDefinition.isReference` is true. The type of this `Element` must conform to the type specified in the associated `ExtensionAttributeDefinition`.
extensionAttributeDefinition: ExtensionAttributeDefinition	Defines the extension attribute for which this value is being provided.

Extensibility XML Schemas

Table 8-11 – Extension XML schema

```
<xsd:element name="extension" type="tExtension"/>
<xsd:complexType name="tExtension">
    <xsd:sequence>
        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="definition" type="xsd:QName"/>
    <xsd:attribute name="mustUnderstand" type="xsd:boolean" use="optional"/>
</xsd:complexType>
```

XML Example

This example shows a Task, defined the BPMN Core, being extended with Inputs and Outputs defined outside of the Core.

Table 8-12 – Example Core XML schema

```
<xsd:schema …>

        …

    <xsd:element name="task" type="tTask"/>
    <xsd:complexType name="tTask">
        <xsd:complexContent>
            <xsd:extension base="tActivity"/>
        </xsd:complexContent>
    </xsd:complexType>

        …

</xsd:schema>
```

Table 8-13 – Example Extension XML schema

```
<xsd:schema …>

        …

    <xsd:group name="dataRequirements">
        <xsd:sequence>
            <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element ref="inputSet" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element ref="outputSet" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:group>

        …

</xsd:schema>
```

Table 8-14 – Sample XML instance

```
<bpmn:definitions id="ID_1" …>

        …

    <bpmn:extension mustUnderstand="true" definition="bpmn:dataRequirements"/>

        …

    <bpmn:task name="Retrieve Customer Record" id="ID_2">
        <bpmn:dataInput name="Order Input" id="ID_3">
            <bpmn:typeDefinition typeRef="bo:Order" id="ID_4"/>
        </bpmn:dataInput>
        <bpmn:dataOutput name="Customer Record Output" id="ID_5">
            <bpmn:typeDefinition typeRef="bo:CustomerRecord" id="ID_6"/>
        </bpmn:dataOutput>
        <bpmn:inputSet name="Inputs" id="ID_7" dataInputRefs="ID_3"/>
        <bpmn:outputSet name="Outputs" id="ID_8" dataOutputRefs="ID_5"/>
    </bpmn:task>

        …

</bpmn:definitions>
```

8.2.4. External Relationships

It is the intention of this specification to cover the basic elements required for the construction of semantically rich and syntactically valid Process models to be used in the description of Processes, Choreographies and business operations in multiple levels of abstraction. As the specification indicates, extension capabilities enable the enrichment of the information described in BPMN and supporting models to be augmented to fulfill particularities of a given usage model. These extensions intention is to extend the semantics of a given BPMN Artifact to provide specialization of intent or meaning.

Process models do not exist in isolation and generally participate in larger, more complex business and system development Processes. The intention of the following specification element is to enable BPMN Artifacts to be integrated in these development Processes via the specification of a non-intrusive identity/relationship model between BPMN Artifacts and elements expressed in any other addressable domain model.

The 'identity/relationship' model it is reduced to the creation of families of typed relationships that enable BPMN and non-BPMN Artifacts to be related in non intrusive manner. By simply defining 'relationship types' that can be associated with elements in the BPMN Artifacts and arbitrary elements in a given addressable domain model, it enables the extension and integration of BPMN models into larger system/development Processes.

It is that these extensions will enable, for example, the linkage of 'derivation' or 'definition' relationships between UML artifacts and BPMN Artifacts in novel ways. So, a UML use case could be related to a Process element in the BPMN specification without affecting the nature of the Artifacts themselves, but enabling different integration models that traverse specialized relationships.

Simply, the model enables the external specification of augmentation relationships between BPMN Artifacts and arbitrary relationship classification models, these external models, via traversing relationships declared in the external definition allow for linkages between BPMN elements and other structured or non-structured metadata definitions.

The UML model for this specification follow a simple extensible pattern as shown below; where named relationships can be established by referencing objects that exist in their given namespaces.

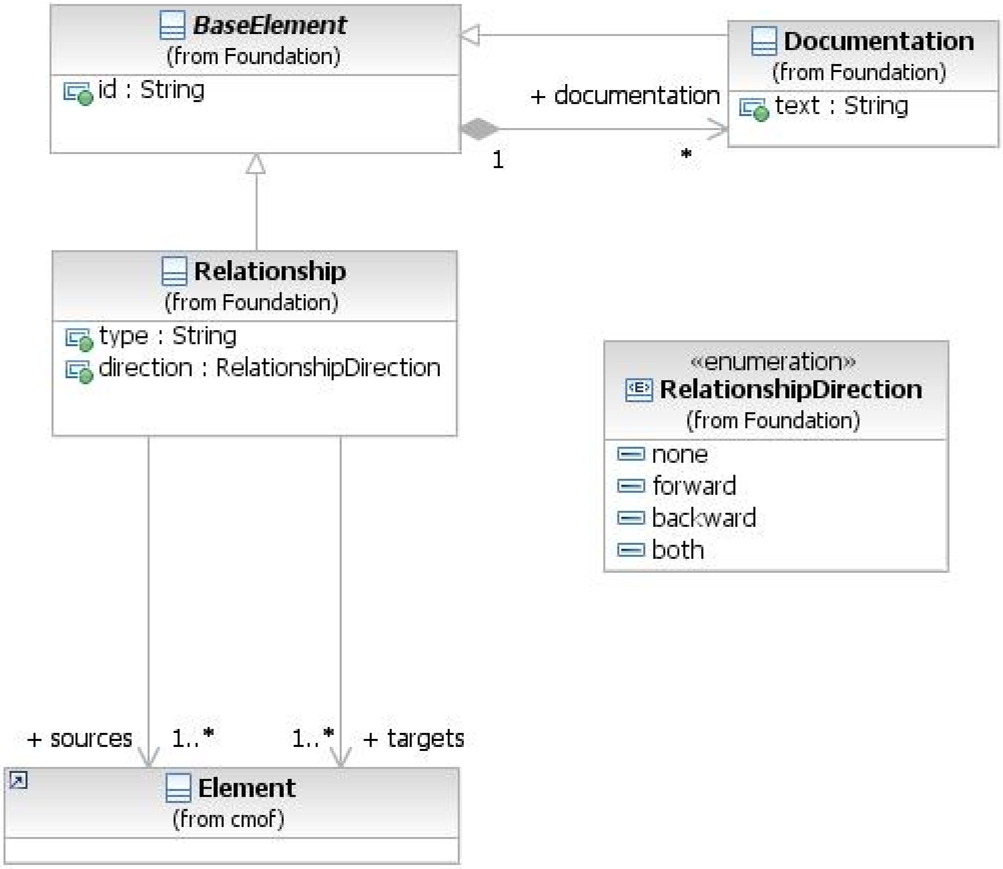


Figure 8-7 – External Relationship Metamodel

The `Relationship` element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 8-15 presents the additional attributes for the `Relationship` element:

Table 8-15 – Relationship attributes

Attribute Name	Description/Usage
type: string	The descriptive name of the element.
direction: RelationshipDirection {none | forward | backward | both}	This attribute specifies the direction of the relationship.
sources: Element [1..*]	This association defines artifacts that are augmented by the relationship.
targets: Element [1..*]	This association defines artifacts used to extend the semantics of the source element(s).

In this manner, you can, for example, create relationships between different artifacts that enable external annotations used for (for example) traceability, derivation, arbitrary classifications, etc.

An example where the ‘reengineer’ relationship is shown between elements in a Visio ™ artifact and a BPMN Artifact:

Table 8-16 – Reengineer XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace=""
    typeLanguage="" id="a123" expressionLanguage=""
    xsi:schemaLocation="http://www.omg.org/bpmn20 Core-Common.xsd"
    xmlns="http://www.omg.org/bpmn20"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:src="http://www.example.org/Processes/Old"
    xmlns:tgt="http://www.example.org/Processes/New">

    <import importType="http://office.microsoft.com/visio" location="OrderConfirmationProcess.vsd"
            namespace="http://www.example.org/Processes/Old"/>
    <import importType="http://www.omg.org/bpmn20" location="OrderConfirmationProcess.xml"
            namespace="http://www.example.org/Processes/New"/>

    <relationship type="reengineered" id="a234" direction="both">
        <documentation>An as-is and to-be relationship. The as-is model is expressed as a Visio diagram.
                The re-engineered process has been split in two and is captured in BPMN 2.0
                format.</documentation>
        <source ref="src:OrderConfirmation"/>
        <target ref="tgt:OrderConfirmation_PartI"/>
        <target ref="tgt:OrderConfirmation_PartII"/>
    </relationship>
</definitions>
```

8.2.5. Root Element

`RootElement` is the abstract super class for all BPMN elements that are contained within `Definitions`. When contained within `Definitions`, these elements have their own defined life-cycle and are not deleted with the deletion of other elements. Examples of concrete `RootElements` include `Collaboration`, `Process`, and `Choreography`. Depending on their use, `RootElements` can be referenced by multiple other elements (i.e., they can be reused). Some `RootElements` may be contained within other elements instead of `Definitions`. This is done to avoid the maintenance overhead of an independent life-cycle. For example, an `EventDefinition` may be contained in a `Process` since it may be only required there. In this case the `EventDefinition` would be dependent on the tool life-cycle of the `Process`.

The `RootElement` element inherits the attributes and model associations of `BaseElement` (see Table 8-5), but does not have any further attributes or model associations.

8.2.6. Foundation Package XML Schemas

Table 8-17 – BaseElement XML schema

```
<xsd:element name="baseElement" type="tBaseElement"/>
<xsd:complexType name="tBaseElement" abstract="true">
    <xsd:sequence>
        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="category" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="baseElementWithMixedContent" type="tBaseElementWithMixedContent"/>
<xsd:complexType name="tBaseElementWithMixedContent" abstract="true" mixed="true">
    <xsd:sequence>
        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="category" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>

<xsd:element name="documentation" type="tDocumentation"/>
<xsd:complexType name="tDocumentation" mixed="true">
    <xsd:sequence>
        <xsd:any namespace="##any" processContents="lax" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
```

Table 8-18 – RootElement XML schema

```
<xsd:element name="rootElement" type="tRootElement"/>
<xsd:complexType name="tRootElement" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType
```

Table 8-19 – Relationship XML schema

```
<xsd:element name="relationship" type="tRelationship"/>
<xsd:complexType name="tRelationship">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="source" type="xsd:QName" minOccurs="1"
                        maxOccurs="unbounded"/>
                <xsd:element name="target" type="xsd:QName" minOccurs="1"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="type" type="xsd:string" use="required"/>
            <xsd:attribute name="direction" type="tRelationshipDirection"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tRelationshipDirection">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="none"/>
        <xsd:enumeration value="forward"/>
        <xsd:enumeration value="backward"/>
        <xsd:enumeration value="both"/>
    </xsd:restriction>
</xsd:simpleType>
```

8.3. Common Elements

The following sections define BPMN elements that may be used in more than one type of diagram (e.g., Process, Collaboration, Conversation, and Choreography).

8.3.1. Artifacts

BPMN provides modelers with the capability of showing additional information about a Process that is not directly related to the Sequence Flow or Message Flow of the Process.

At this point, BPMN provides three (3) standard Artifacts: Associations, Groups, and a Text Annotations. Additional Artifacts may be added to the BPMN specification in later versions. A modeler or modeling tool may extend a BPMN diagram and add new types of Artifacts to a Diagram. Any new Artifact must follow the Sequence Flow and Message Flow connection rules (listed below). Associations can be used to link Artifacts to Flow Objects (see page 87).

Figure 8-8 shows the Artifacts class diagram. When an Artifact is defined it is contained within a Collaboration or a FlowElementsContainer (a Process or Choreography).

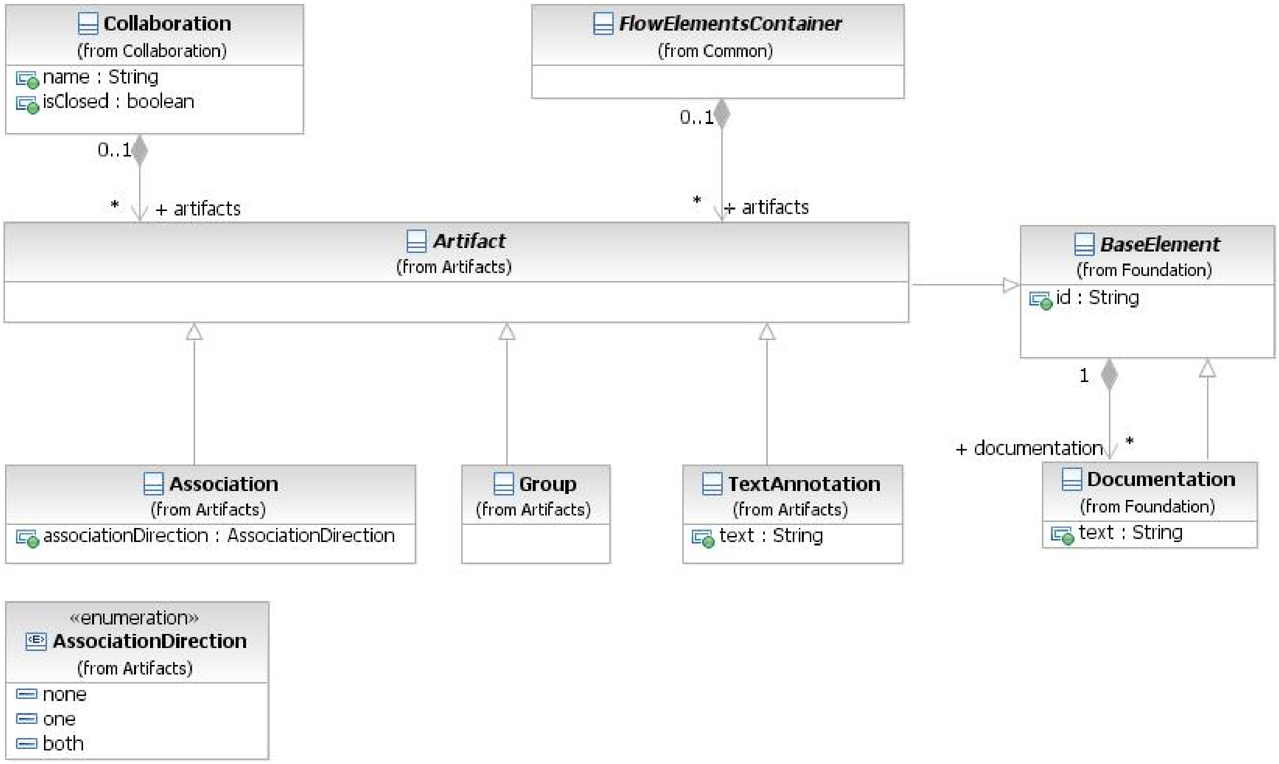


Figure 8-8 – Artifacts Metamodel

Common Artifact Definitions

The following sections provide definitions that a common to all Artifacts.

Artifact Sequence Flow Connections

See “Sequence Flow Rules,” on page 64 for the entire set of objects and how they may be source or targets of Sequence Flow.

- An Artifact MUST NOT be a target for Sequence Flow.
- An Artifact MUST NOT be a source for Sequence Flow.

Artifact Message Flow Connections

See “Message Flow Rules,” on page 65 for the entire set of objects and how they may be source or targets of Message Flow.

- An Artifact MUST NOT be a target for Message Flow.
- An Artifact MUST NOT be a source for Message Flow.

Association

An Association is used to associate information and Artifacts with *Flow Objects*. Text and graphical non-*Flow Objects* can be associated with the *Flow Objects* and Flow. An Association is also used to show the Activity used for *compensation*. More information about *compensation* can be found page on 314.

- ◆ An Association is line that MUST be drawn with a dotted single line (see Figure 8-9).
    - ◆ The use of text, color, size, and lines for an Association MUST follow the rules defined in section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

....

Figure 8-9 – An Association

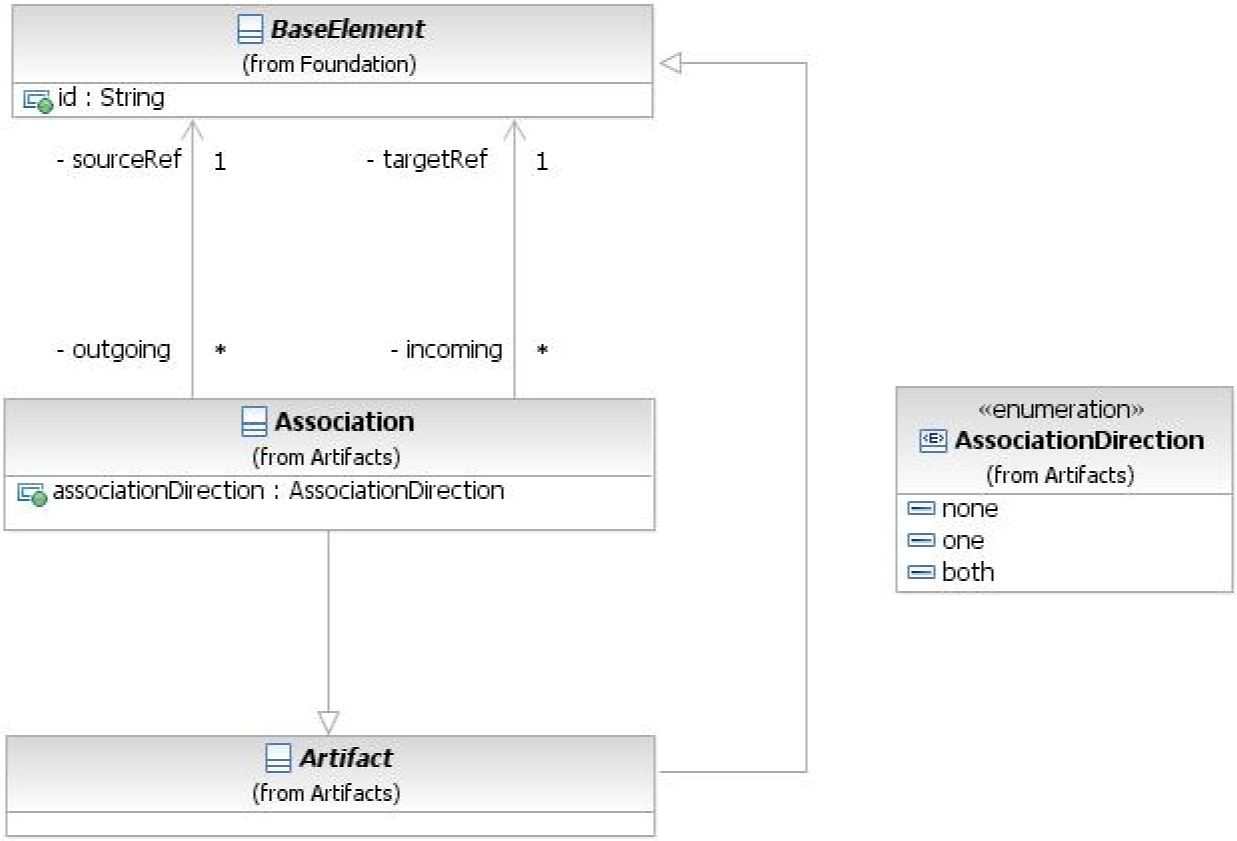


Figure 8-10 – The Association Class Diagram

If there is a reason to put directionality on the Association then:

- ◆ A line arrowhead MAY be added to the Association line (see Figure 8-11).
    - ◆ The directionality of the Association can be in one (1) direction or in both directions.

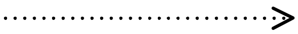


Figure 8-11 – A Directional Association

Note that directional Associations were used in BPMN 1.2 to show how Data Objects were inputs or outputs to Activities. In BPMN 2.0, a Data Association connector is used to show inputs and outputs (see page 228). A Data Association uses the same notation as a directed Association (as in Figure 8-11, above).

An Association is used to connect user-defined text (an Annotation) with a *Flow Object* (see Figure 8-12).

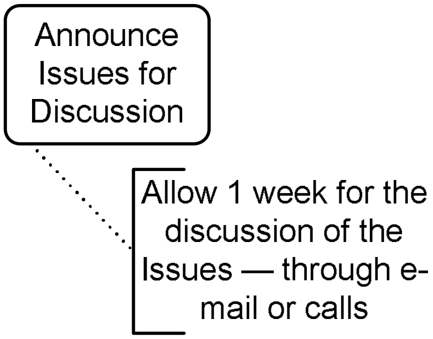

Announce Issues for Discussion
Allow 1 week for the discussion of the Issues — through e-mail or calls

Figure 8-12 – An Association of Text Annotation

The Association element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-20 presents the additional attributes and model associations for an Association:

Table 8-20 –Association attributes and model associations

Attributes	**Description**

associationDirection:
AssociationDirection = None
{None | One | Both}
	associationDirection is an attribute that defines whether or not the Association shows any directionality with an arrowhead. The default is None (no arrowhead). A value of One means that the arrowhead SHALL be at the Target Object. A value of Both means that there SHALL be an arrowhead at both ends of the Association line.

sourceRef: BaseElement	The BaseElement that the Association is connecting from.
targetRef: BaseElement	The BaseElement that the Association is connecting to.

Group

The Group object is an Artifact that provides a visual mechanism to group elements of a diagram informally. The grouping is tied to the Category supporting element (which is an attribute of all BPMN elements). That is, a Group is a visual depiction of a single Category. The graphical elements within the

Group will be assigned the Category of the Group. (Note -- Categories can be highlighted through other mechanisms, such as color, as defined by a modeler or a modeling tool).

- A Group is a rounded corner rectangle that MUST be drawn with a solid dashed line (as seen in Figure 8-13).
  - The use of text, color, size, and lines for a Group MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

Figure 8-13 – A Group Artifact

As an Artifact, a Group is not an Activity or any Flow Object, and, therefore, cannot connect to Sequence Flow or Message Flow. In addition, Groups are not constrained by restrictions of Pools and Lanes. This means that a Group can stretch across the boundaries of a Pool to surround Diagram elements (see Figure 8-14), often to identify Activities that exist within a distributed business-to-business transaction.

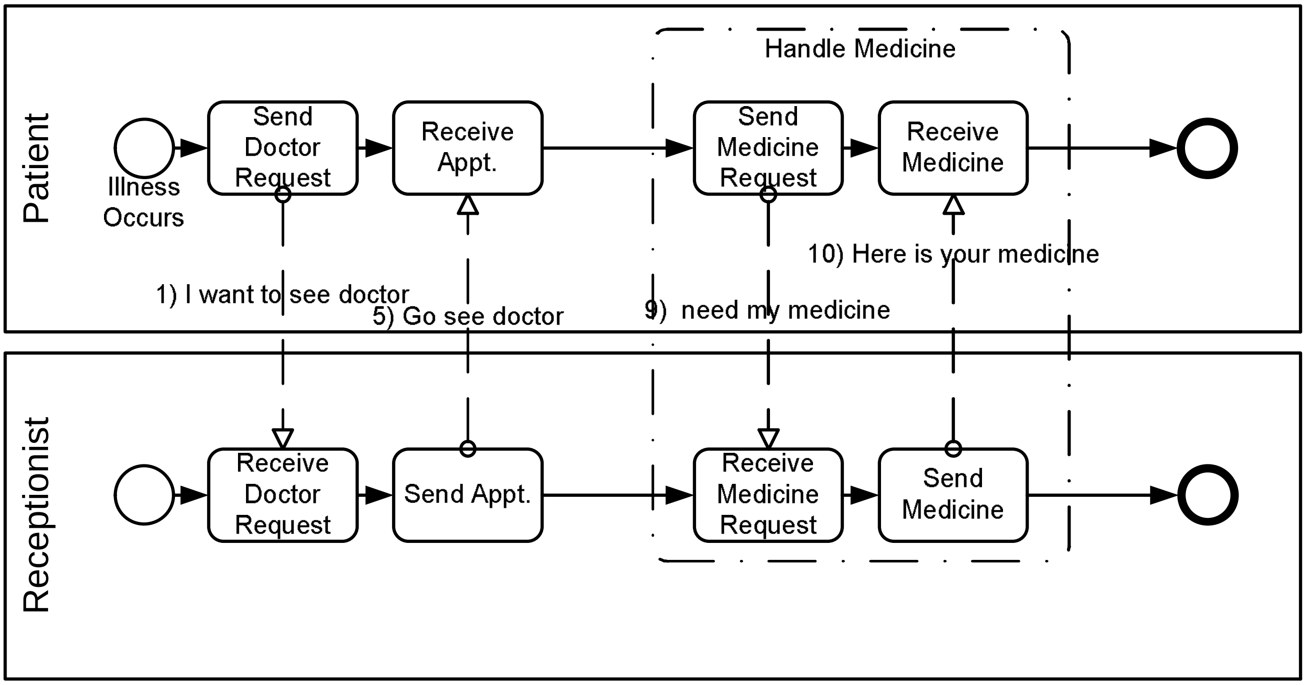


Figure 8-14 – A Group around Activities in different Pools

Groups are often used to highlight certain sections of a Diagram without adding additional constraints for performance--as a Sub-Process would. The highlighted (grouped) section of the Diagram can be separated for reporting and analysis purposes. Groups do not affect the flow of the Process.

Figure 8-15 shows the Group class diagram.

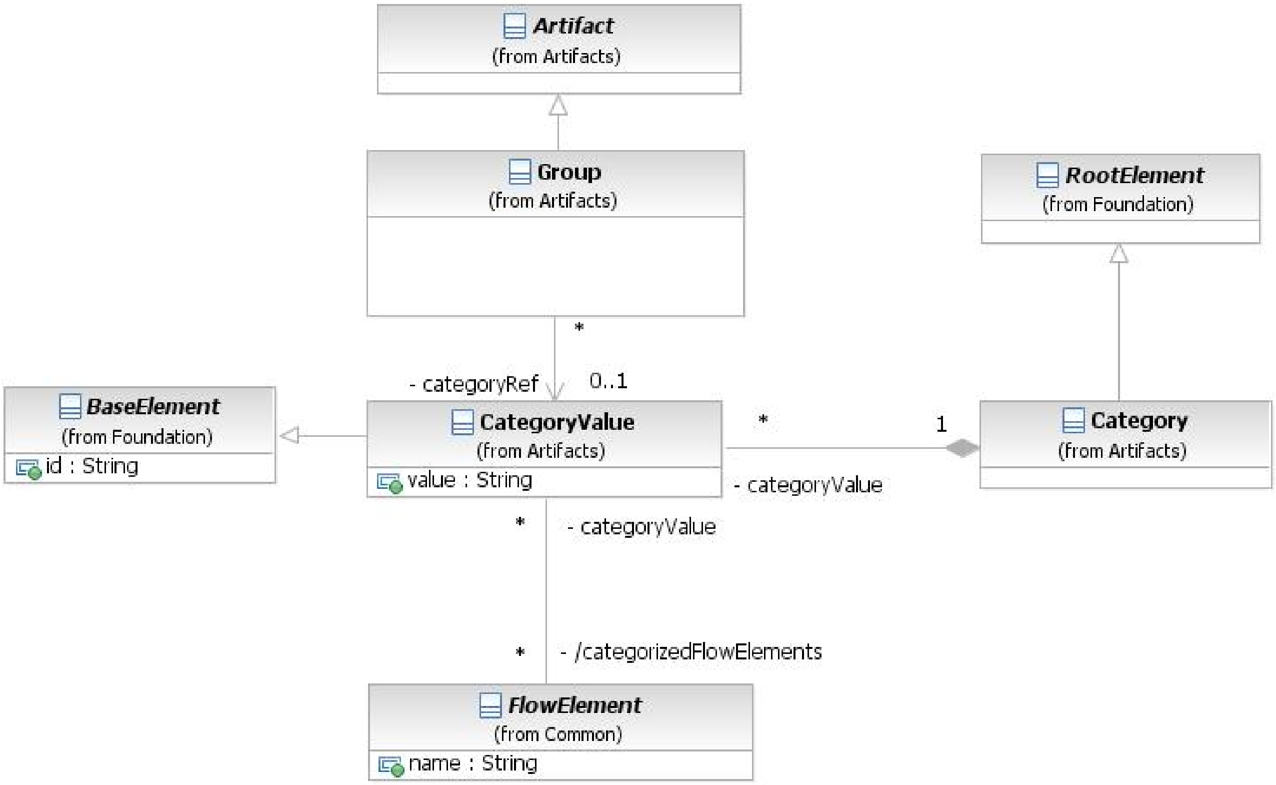


Figure 8-15 – The Group class diagram

The Group element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-21 presents the additional model associations for a Group:

Table 8-21 –Group model associations

Attributes	Description
categoryRef: Category [0..1]	The categoryRef attribute specifies the Category that the Group represents (Further details about the definition of a Category can be found on page 92). The name of the Category provides the label for the Group. The graphical elements within the boundaries of the Group will be assigned the Category.

Category

Categories, which have user-defined semantics, can be used for documentation or analysis purposes. For example, FlowElements can be categorized has being customer oriented vs. support oriented. For example, the cost and time of each Category of each Activity can be calculated.

Groups are one way in which Categories of objects can be visually displayed on the diagram. That is, a Group is a visual depiction of a single Category. The graphical elements within the Group will be assigned the Category of the Group. The Category name appears on the diagram as the Group label. (Note -- Categories can be highlighted through other mechanisms, such as color, as defined by a modeler or a modeling tool). A single Category can be used for multiple Groups in a diagram.

The Category element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-22 displays the additional model associations of the Category element.

Table 8-22 –Category model associations

Attributes	Description
categoryValue: CategoryValue [0..*]	The categoryValue attribute specifies one or more values of the Category. For example, the Category is "Region" then this Category could specifies values like "North", "South", "West" and "East"

The Category element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-23 displays the attributes and model associations of the CategoryValue element.

Table 8-23 –CategoryValue attributes and model associations

Attributes	Description
value: string	This attribute provides the value of the CategoryValue element.
category: Category [0..1]	The category attribute specifies the Category representing the Category as such and contains the CategoryValue (Further details about the definition of a Category can be found on page 92).
categorizedFlowElements: FlowElement [0..*]	The FlowElements attribute identifies all of the elements (e.g., Events, Activities, Gateways, and Artifacts) that are within the boundaries of the Group.

Text Annotation

Text Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN Diagram.

- A Text Annotation is an open rectangle that MUST be drawn with a solid single line (as seen in Figure 8-16).
  - The use of text, color, size, and lines for a Text Annotation MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63.

The Text Annotation object can be connected to a specific object on the Diagram with an Association, but do not affect the flow of the Process. Text associated with the Annotation can be placed within the bounds of the open rectangle.

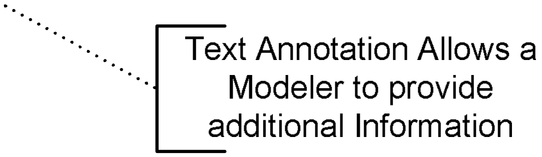


Figure 8-16 – A Text Annotation

The Text Annotation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-24 presents the additional attributes for a Text Annotation:

Table 8-24 –Text Annotation attributes

Attributes	Description
text: string	Text is an attribute that is text that the modeler wishes to communicate to the reader of the Diagram.

XML Schema for Artifacts

Table 8-25 – Artifact XML schema

```
<xsd:element name="artifact" type="tArtifact"/>
<xsd:complexType name="tArtifact" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-26 – Association XML schema

```
<xsd:element name="association" type="tAssociation" substitutionGroup="artifact"/>
<xsd:complexType name="tAssociation">
    <xsd:complexContent>
        <xsd:extension base="tArtifact">
            <xsd:attribute name="sourceRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="targetRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="associationDirection" type="tAssociationDirection" default="none"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tAssociationDirection">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="none"/>
        <xsd:enumeration value="one"/>
        <xsd:enumeration value="both"/>
        </xsd:restriction>
</xsd:simpleType>
```

Table 8-27 – Category XML schema

```
<xsd:element name="category" type="tCategory" substitutionGroup="rootElement"/>
<xsd:complexType name="tCategory">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element ref="categoryValue" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-28 – Group XML schema

```
<xsd:element name="group" type="tGroup" substitutionGroup="artifact"/>
<xsd:complexType name="tGroup">
      <xsd:complexContent>
            <xsd:extension base="tArtifact">
                  <xsd:attribute name="categoryRef" type="xsd:QName" use="optional"/>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 8-29 – Text Annotation XML schema

```
<xsd:element name="textAnnotation" type="tTextAnnotation" substitutionGroup="artifact"/>
<xsd:complexType name="tTextAnnotation">
        <xsd:complexContent>
                <xsd:extension base="tArtifact">
                        <xsd:sequence>
                              <xsd:element ref="text" minOccurs="0" maxOccurs="1"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:element name="text" type="tText"/>
<xsd:complexType name="tText" mixed="true">
     <xsd:sequence>
          <xsd:any namespace="##any" processContents="lax" minOccurs="0"/>
     </xsd:sequence>
</xsd:complexType>
```

8.3.2. Callable Element

`CallableElement` is the abstract super class of all `Activities` that have been defined outside of a `Process` or `Choreography` but which can be called (or reused) from within a `Process` or `Choreography`. It may reference `Interfaces` that define the service operations that it provides.

`CallableElements` are `RootElements`, which can be imported and used in other `Definitions`. When `CallableElements` (e.g., `Process`) are defined, they are contained within `Definitions`.

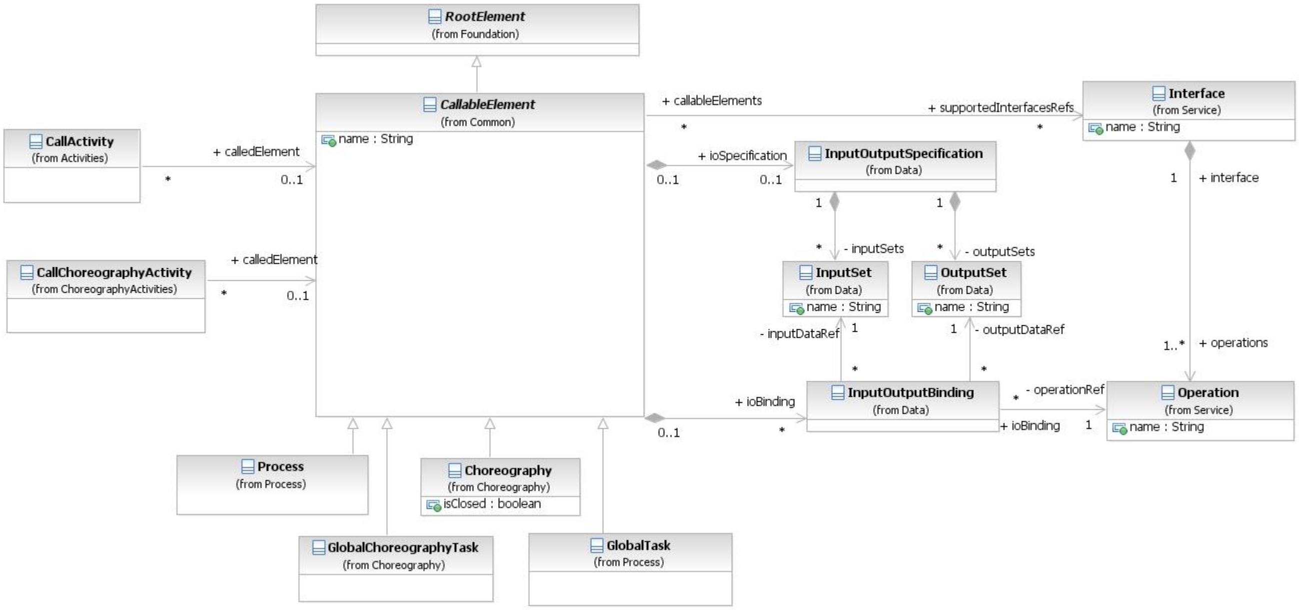


Figure 8-17 – CallableElement class diagram

The CallableElement inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-30 presents the additional attributes and model associations of the CallableElement:

Table 8-30 – CallableElement attributes and model associations

Attribute Name	Description/Usage
name: string [0..1]	The descriptive name of the element.
supportedInterfacesRefs: Interface [0..*]	The Interfaces describing the external behavior provided by this element.
ioSpecification: InputOutputSpecification [0..1]	The InputOutputSpecification defines the *inputs* and *outputs* and the InputSets and OutputSets for the Activity.
ioBinding: InputOutputBinding [0..*]	The InputOutputBinding defines a combination of one InputSet and one OutputSet in order to bind this to an operation defined in an interface.

When a CallableElement is exposed as a Service, it has to define one or more InputOutputBinding elements. An InputOutputBinding element binds one *Input* and one *Output* of the InputOutputSpecification to an Operation of a Service Interface. Table 8-31 presents the additional model associations of the InputOutputBinding:

Table 8-31 – InputOutputBinding model associations

Attribute Name	Description/Usage
inputDataRef: DataInput	A reference to one specific DataInput defined as part of the InputOutputSpecification of the Activity.
outputDataRef: DataOutput	A reference to one specific DataOutput defined as part of the InputOutputSpecification of the Activity.
operationRef: Operation	A reference to one specific Operation defined as part of the Interface of the Activity.

8.3.3. Correlation

The concept of *Correlation* facilitates the association of a Message to a Send Task or Receive Task involved in a Conversation, a mechanism BPMN refers to as *instance routing*. It is a particular useful concept where there is no infrastructure support for *instance* routing. Note that this association can be viewed at multiple levels, namely the Conversation, Choreography, and Process level. However, the actual correlation happens during runtime (e.g. at the Process level). *Correlations* describe a set of predicates on a Message (generally on the application payload) that need to be satisfied in order for that Message to be associated to a distinct Send Task or Receive Task. By the same token, each Send Task and each Receive Task participates in one or many Conversations. Furthermore, it identifies the Message it sends or receives and thereby establishes the relationship to one (or many) CorrelationKeys.

There are two, non-exclusive correlation mechanisms in place:

- In plain, key-based correlation, Messages that are exchanged within a Conversation are logically correlated by means of a joint CorrelationKey. That is, any Message that is sent or received within this Conversation needs to carry the value of the CorrelationKey *instance* within its payload. A CorrelationKey basically defines a (composite) key. The first Message that is initially sent or received initializes the CorrelationKey *instance*, i.e. assigns values to its CorrelationProperty *instances* which are the fields (partial keys) of the CorrelationKey. Follow-up Messages will have to match the CorrelationKey *instance* to be dispatched to this particular Conversation. As a Conversation may comprise different Messages that may be differently structured, each CorrelationProperty comes with as many extraction rules (CorrelationPropertyRetrievalExpression) for the respective partial key as there are different Messages.
- In context-based correlation, the Process context (i.e., its Data Objects and Properties) may dynamically influence the matching criterion. That is, a CorrelationKey may be complemented by a Process-specific CorrelationSubscription. A CorrelationSubscription aggregates as many CorrelationPropertyBindings as there are CorrelationProperties in the CorrelationKey. A CorrelationPropertyBinding relates to a specific CorrelationProperty and also links to a FormalExpression which denotes a dynamic extraction rule atop the Process context. At runtime, the CorrelationKey instance for a particular Conversation is populated (and dynamically updated) from the Process context using these FormalExpressions. In that sense, changes in the Process context may alter the correlation condition.

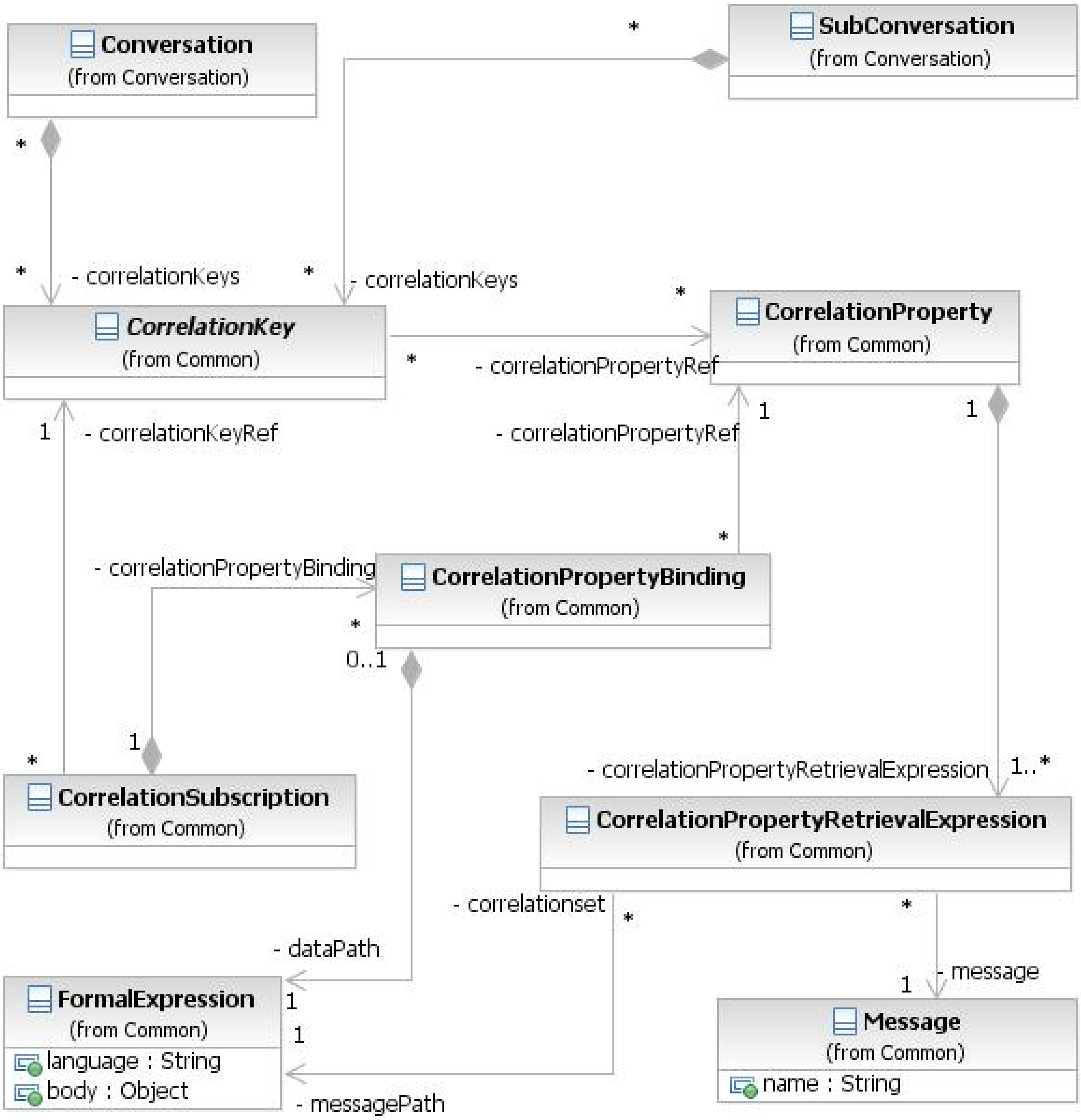


Figure 8-18 – The Correlation Class Diagram

CorrelationKey

A `CorrelationKey` represents a composite key out of one (1) or many `CorrelationProperties` which essentially specify extraction `Expressions` atop `Messages`. As a result, each `CorrelationProperty` acts as a partial key for the *correlation*. For each `Message` that is received within a particular `Conversation`, the `CorrelationProperties` need to provide a `CorrelationPropertyRetrievalExpression` which references a `FormalExpression` to the

Message payload. That is, for each Message (that is used in a Conversation) there is an Expression which extracts portions of the respective Message's payload.

The CorrelationKey element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-32 displays the additional model associations of the CorrelationKey element.

Table 8-32 – CorrelationKey model associations

Attribute Name	Description/Usage
conversation: Conversation	A reference to the specific Conversation this CorrelationKey applies to.
correlationPropertyRef: CorrelationProperty [0..*]	The CorrelationProperties, representing the partial keys of this CorrelationKey

Key-based Correlation

Key-based *correlation* is a simple and efficient form of *correlation*, where one or more keys are used to identify a Conversation. Any incoming Message can be matched against the CorrelationKey by extracting the CorrelationProperties from the Message according to the corresponding CorrelationPropertyRetrievalExpression and comparing the resulting composite key with the CorrelationKey instance for this Conversation. The idea is to use a joint Conversation "token" which is used (passed to and received from) and *outgoing* and *incoming* Message. Messages will be routed to the Conversation whose extracted composite key matches the respective CorrelationKey *instance*.

At runtime the first Send Task or Receive Task in a Conversation populates the CorrelationKey instance by extracting the values of the CorrelationProperties according to the CorrelationPropertyRetrievalExpression from the initially sent or received Message. Later in the Conversation, this CorrelationKey instance is used for the described matching procedure where from incoming Messages a composite key is extracted and used to identify the associated Conversation.

The CorrelationProperty element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-33 displays the additional model associations of the CorrelationProperty element.

Table 8-33 – CorrelationProperty model associations

Attribute Name	Description/Usage
correlationPropertyRetrievalExpression: CorrelationPropertyRetrievalExpression [1..*]	The CorrelationPropertyRetrievalExpressions for this CorrelationProperty, representing the associations of FormalExpressions (extraction paths) to specific Messages occurring in this Conversation.

The CorrelationPropertyRetrievalExpression element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-34 displays the additional model associations of the CorrelationPropertyRetrievalExpression element.

Table 8-34 – CorrelationPropertyRetrievalExpression model associations

Attribute Name	Description/Usage
messagePath: FormalExpression	The FormalExpression that defines how to extract a CorrelationProperty from the Message payload
message: Message	The specific Message the FormalExpression extracts the CorrelationProperty from.

Context-based Correlation

Context-based *correlation* is a more expressive form of *correlation* on top of key-based *correlation*. In addition to implicitly populating the CorrelationKey *instance* from the first sent or received Message, another mechanism relates the CorrelationKey to the Process context. That is, a Process may provide a CorrelationSubscription which acts as the Process-specific counterpart to a specific CorrelationKey. In this way, a Conversation may additionally refer to explicitly updateable Process context data to determine whether or not a Message shall be received. At runtime, the CorrelationKey instance holds a composite key that is dynamically calculated from the Process context and automatically updated whenever the underlying Data Objects or Properties change.

CorrelationPropertyBindings represent the partial keys of a CorrelationSubscription where each relates to a specific CorrelationProperty in the associated CorrelationKey. A FormalExpression defines how that CorrelationProperty *instance* is populated and updated at runtime from the Process context (i.e., its Data Objects and Properties).

The CorrelationSubscription element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-35 displays the additional model associations of the CorrelationSubscription element.

Table 8-35 – CorrelationSubscription model associations

Attribute Name	Description/Usage
Process: Process	The Process that this CorrelationSubscription belongs to.
correlationKeyRef: CorrelationKey	The CorrelationKey this CorrelationSubscription refers to.
correlationPropertyBinding: CorrelationPropertyBinding [0..*]	The bindings to specific CorrelationProperties and FormalExpressions (extraction rules atop the Process context).

The CorrelationPropertyBinding element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-36 displays the additional model associations of the CorrelationPropertyBinding element.

Table 8-36 – CorrelationPropertyBinding model associations

Attribute Name	Description/Usage
dataPath: FormalExpression	The FormalExpression that defines the extraction rule atop the Process context.
correlationPropertyRef: CorrelationProperty	The specific CorrelationProperty, this CorrelationPropertyBinding refers to.

At runtime, the correlation mechanism works as follows: When a Process instance is created the CorrelationKey instances of all Conversations are initialized with some initial values that specify to correlate *any* incoming Message for these Conversations. A SubscriptionProperty is updated whenever any of the Data Objects or Properties changes that are referenced from the respective FormalExpression. As a result, incoming Messages are matched against the now populated CorrelationKey instance. Later in the Process run, the SubscriptionProperties may, again, change and implicitly change the correlation criterion. Alternatively, the established mechanism of having the first Send Task or Receive Task populate the CorrelationKey *instance* applies.

XML Schema for Correlation

Table 8-37 – Correlation Key XML schema

```
<xsd:element name="correlationKey" type="tCorrelationKey"/>
<xsd:complexType name="tCorrelationKey">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="correlationPropertyRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-38 – Correlation Property XML schema

```
<xsd:element name="correlationProperty" type="tCorrelationProperty"/>
<xsd:complexType name="tCorrelationProperty">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element ref="correlationPropertyRetrievalExpression" minOccurs="1"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-39 – Correlation Property Binding XML schema

```
<xsd:element name="correlationPropertyBinding" type="tCorrelationPropertyBinding"/>
<xsd:complexType name="tCorrelationPropertyBinding">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="dataPath" type="tFormalExpression" minOccurs="1"
                        maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="correlationPropertyRef" type="xsd:QName" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-40 – Correlation Property Retrieval Expression XML schema

```
<xsd:element name="correlationPropertyRetrievalExpression"
        type="tCorrelationPropertyRetrievalExpression"/>
<xsd:complexType name="tCorrelationPropertyRetrievalExpression">
        <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="messagePath" type="tFormalExpression" minOccurs="1"
                        maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="messageRef" type="xsd:QName" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-41 – Correlation Subscription XML schema

```
<xsd:element name="correlationSubscription" type="tCorrelationSubscription"/>
<xsd:complexType name=" tCorrelationSubscription ">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="process" type="xsd:QName" use="required"/>
                <xsd:element ref="correlationKeyRef" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="correlationPropertyBinding" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

8.3.4. Conversation Association

A ConversationAssociation is used within Collaborations and Choreographies to apply a reusable Conversation to Message Flow of those diagrams.

A ConversationAssociation is used when a diagram references a Conversation to provide Message correlation information and/or to logically group Message Flow. There are two (2) usages of ConversationAssociation. It is used when:

- A Collaboration references a reusable Conversation. The Message Flow of the Collaboration are grouped by the ConversationAssociation. A tool can use the correlationKey of the Conversation (also referenced) or allow the user to group the Message Flow of the Collaboration.
- A Choreography references a reusable Conversation. The Message Flow of the Choreography are grouped by the ConversationAssociation. A tool can use the correlationKey of the Conversation (also referenced) or allow the user to group the Message Flow of the Choreography.

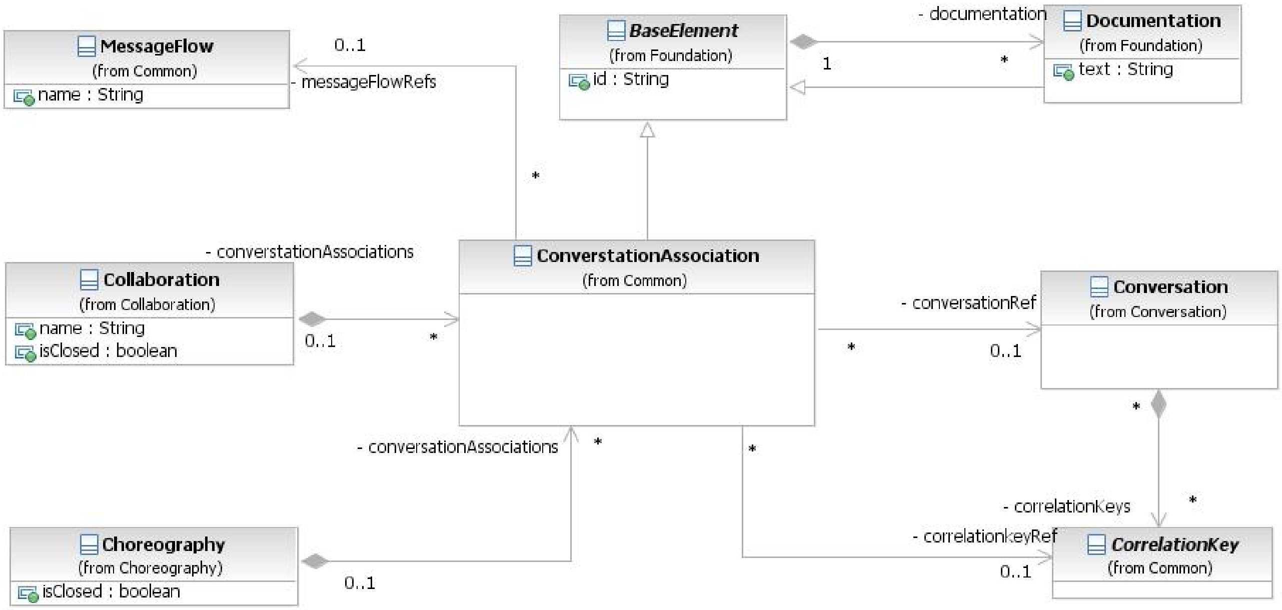


Figure 8-19 – The ConversationAssociation class diagram

The ConversationAssociation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 9-1 presents the additional model associations for the ConversationAssociation element:

Table 8-42 – ConversationAssociation Model Associations

Attribute Name	Description/Usage
conversationRef: Conversation [0..1]	This attribute references the Conversation that is being applied to the Collaboration or Choreography.
correlationKeyRef: Conversation [0..1]	This attribute references a correlationKey in the referenced Conversation that is being applied to the Collaboration or Choreography. This is optional since a CorrelationKey may not be available, but it is necessary when the Conversation contains more than one CorrelationKey.
messageFlowRefs: Message Flow [0..*]	The messageFlowRefs are used to identify the Message Flow within the Collaboration or Choreography that are to be grouped by the referenced Conversation. This grouping can be done automatically through the referenced CorrelationKey of the Conversation (matching the CorrelationKey to the Messages of the Message Flow) or done through user selection if a CorrelationKey has not been defined or referenced.

8.3.5. Error

An Error represents the content of an Error Event or the Fault of a failed Operation. A ItemDefinition is used to specify the structure of the Error. An Error is generated when there is a critical problem in the processing of an Activity or when the execution of an Operation failed.

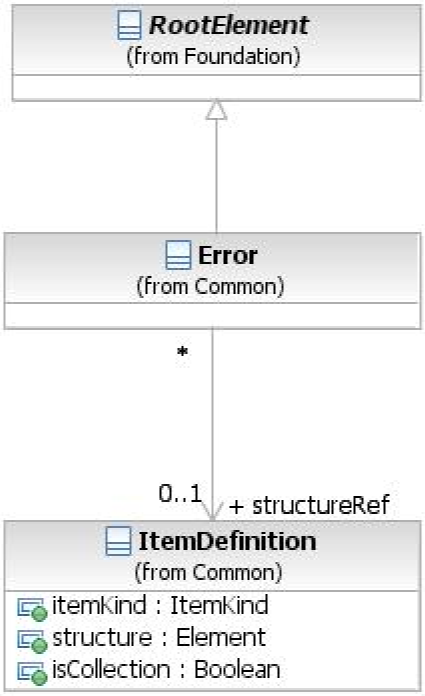


Figure 8-20 – Error class diagram

The Error element inherits the attributes and model associations of BaseElement (see Table 8-5), through its relationship to RootElement. Table 8-43 presents the additional model associations of the Error element:

Table 8-43 – Error model associations

Attribute Name	Description/Usage
structureRef : ItemDefinition [0..1]	An ItemDefinition is used to define the “payload” of the Error.

8.3.6. Events

An Event is something that “happens” during the course of a Process. These Events affect the flow of the Process and usually have a cause or an impact. The term “event” is general enough to cover many things in a Process. The start of an Activity, the end of an Activity, the change of state of a document, a Message that arrives, etc., all could be considered Events. However, BPMN has restricted the use of Events to include only those types of Events that will affect the sequence or timing of Activities of a Process.

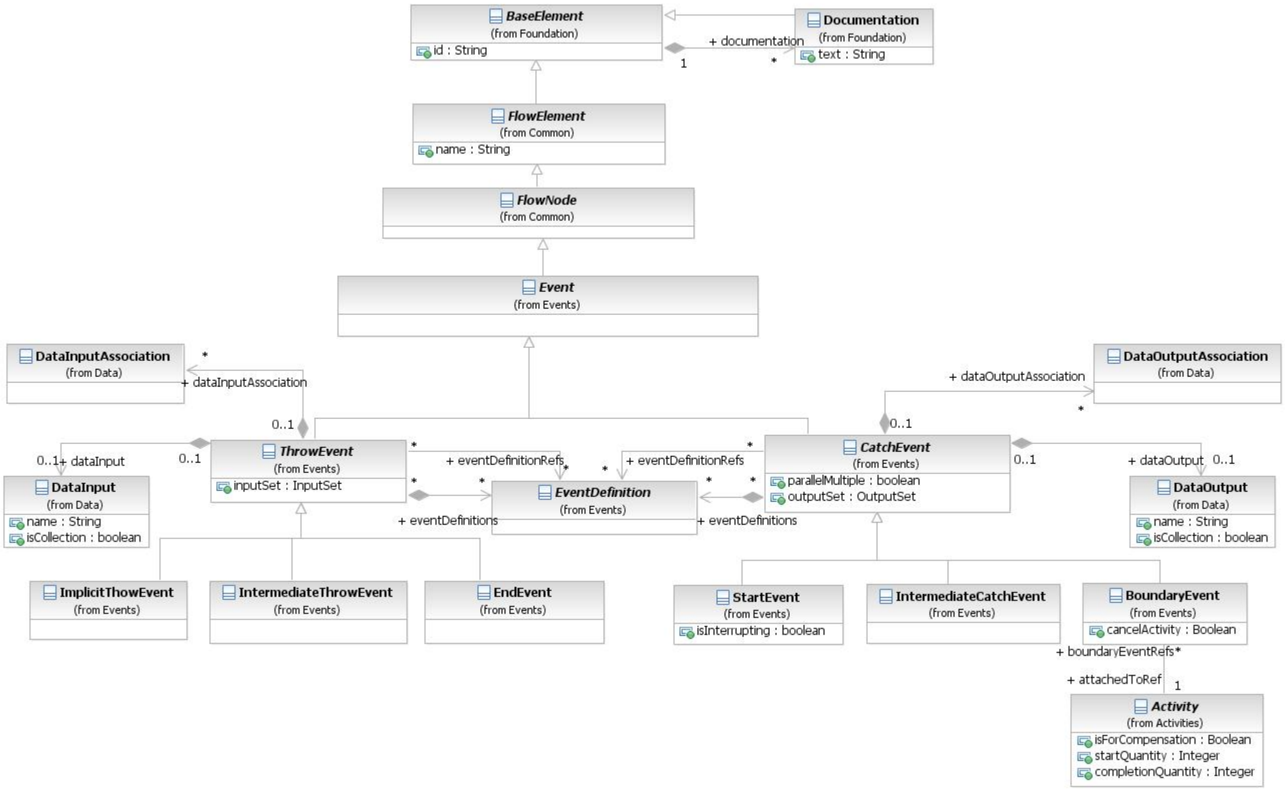


Figure 8-21 – Event class diagram

The Event element inherits the attributes and model associations of Flow Element (see Table 8-45), but adds no additional attributes or model associations:

The details for the types of Events (Start, Intermediate, and End) are defined in the Section "Event Definitions" on page 266.

8.3.7. Expressions

The Expression class is used to specify an expression using natural-language text. These expressions are not executable. The natural language text is captured using the documentation attribute, inherited from BaseElement.

Expression inherits the attributes and model associations of BaseElement (see Table 8-5), but adds no additional attributes or model associations.

Expressions are used in many places within BPMN to extract information from the different elements, normally data elements. The most common usage is when modeling decisions, where conditional expressions are used to direct the flow along specific paths based on some criteria.

BPMN supports underspecified expressions, where the logic is captured as natural-language descriptive text. It also supports formal expressions, where the logic is captured in an executable form using a specified expression language.

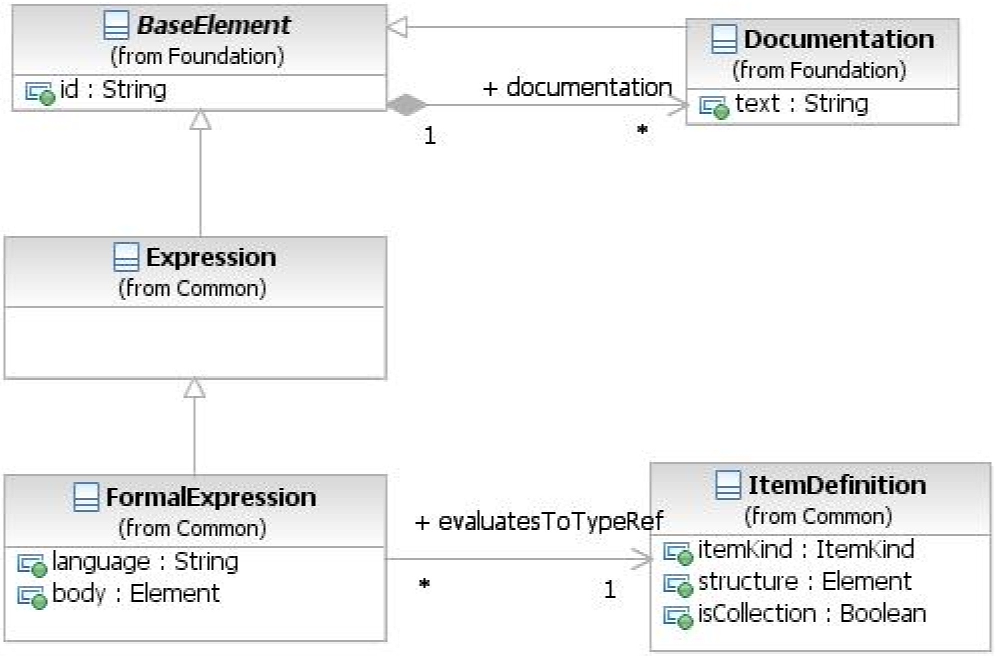


Figure 8-22 – Expression class diagram

Expression

The `Expression` class is used to specify an expression using natural-language text. These expressions are not executable and are considered underspecified.

The definition of an expression can be done in two ways: it can be contained where it is used, or it can be defined at the `Process` level and then referenced where it is used.

The `Expression` element inherits the attributes and model associations of `BaseElement` (see Table 8-5), but does not have any additional attributes or model associations.

Formal Expression

The `FormalExpression` class is used to specify an executable expression using a specified expression language. A natural-language description of the expression can also be specified, in addition to the formal specification.

The default expression language for all expressions is specified in the `Definitions` element, using the `expressionLanguage` attribute. It can also be overridden on each individual `FormalExpression` using the same attribute.

The `FormalExpression` element inherits the attributes and model associations of `BaseElement` (see Table 8-5), through the `Expression` element. Table 8-44 presents the additional attributes and model associations of the `FormalExpression`:

Table 8-44 – FormalExpression attributes and model associations

Attribute Name	Description/Usage
language: string [0..1]	Overrides the expression language specified in the `Definitions`.

body: Element	The body of the `expression`.
	Note that this attribute is not relevant when the XML Schema is used for interchange. Instead, the `FormalExpression` complex type supports mixed content. The body of the `Expression` would be specified as element content. For example:
	`<formalExpression id=“ID_2">`
	   `count(../dataObject[id="CustomerRecord_1"]/emailAddress) > 0`
	   `<evaluatesToType id="ID_3" typeRef=“xsd:boolean"/>`
	`</formalExpression>`

evaluatesToTypeRef: ItemDefinition	The type of object that this `Expression` returns when evaluated. For example, *conditional* `Expressions` evaluate to a *boolean*.

8.3.8. Flow Element

FlowElement is the abstract super class for all elements that can appear in a Process flow, which are FlowNodes (see page 133, which consist of Activities (see page 159), Choreography Activities (see page 349) Gateways (see page 110), and Events (see page 103)), Data Objects (see page 213), Data Associations (see page 228), and Sequence Flow (see page 129),

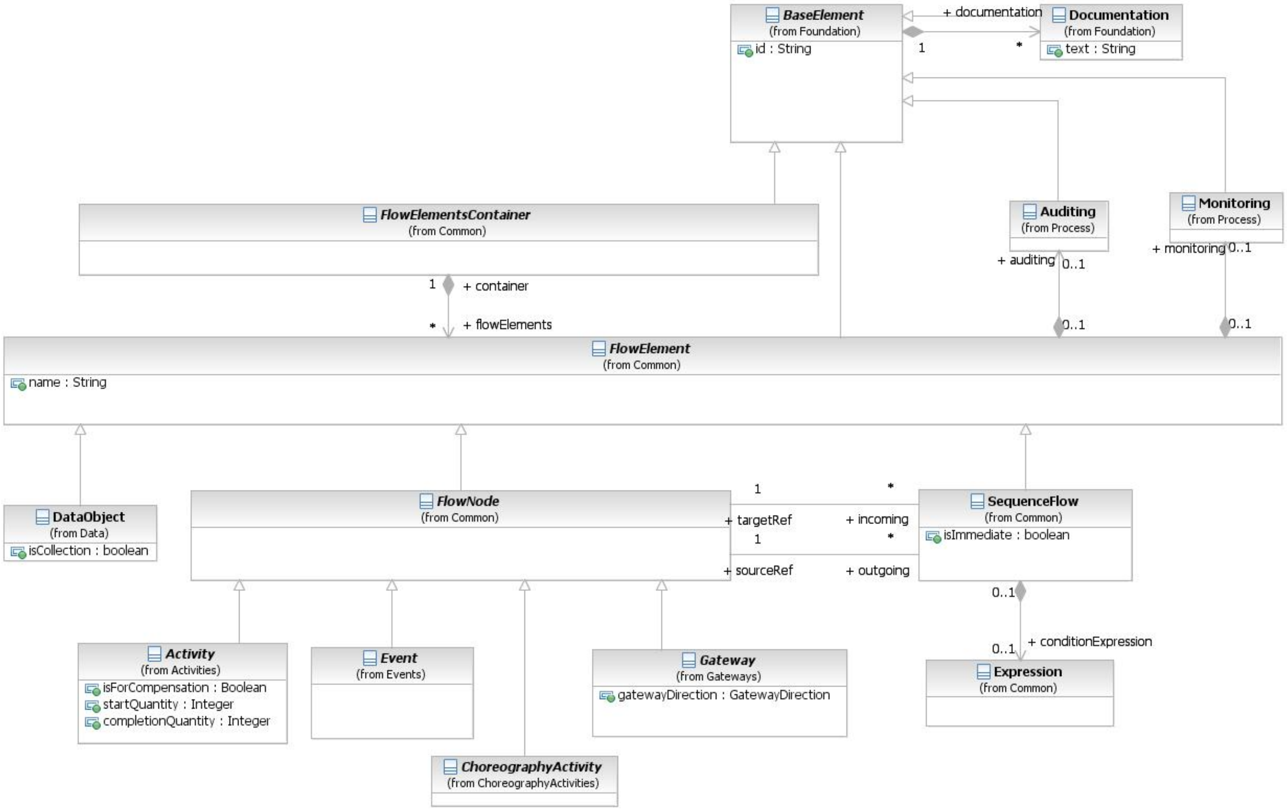


Figure 8-23 – FlowElement class diagram

The FlowElement element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-45 presents the additional attributes and model associations of the FlowElement element:

Table 8-45 – FlowElement attributes and model associations

Attribute Name	Description/Usage
name: string [0..1]	The descriptive name of the element.
auditing: Auditing [0..1]	A hook for specifying audit related properties. Auditing can only be defined for a Process.
monitoring: Monitoring [0..1]	A hook for specifying monitoring related properties. Monitoring can only be defined for a Process.

8.3.9. Flow Elements Container

FlowElementsContainer is an abstract super class for BPMN diagrams (or views) and defines the superset of elements that are contained in those diagrams. Basically, a FlowElementsContainer contains FlowElements, which are Events (see page 103), Gateways (see page 110), Sequence Flow (see page 129), Activities (see page 159), and Choreography Activities (see page 349).

There are four (4) types of FlowElementsContainers (see Figure 8-24): Process, Sub-Process, Choreography, and Choreography Sub-Process.

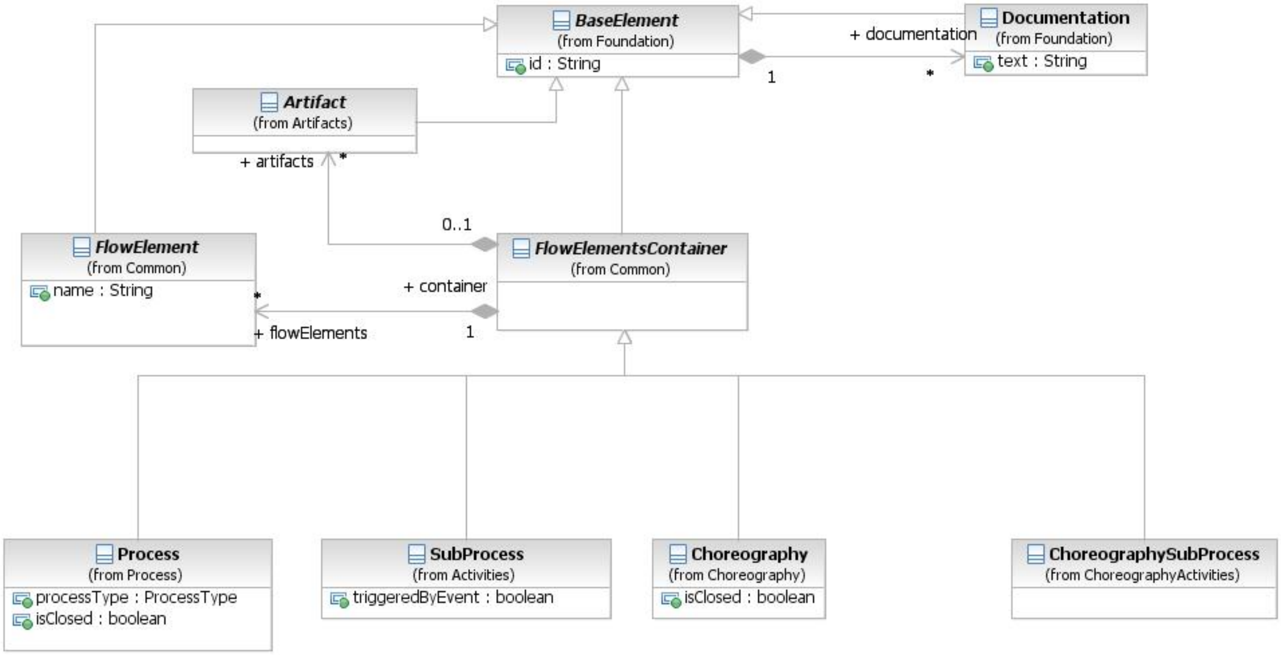


Figure 8-24 – FlowElementContainers class diagram

The FlowElementsContainer element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-46 presents the additional model associations of the FlowElementsContainer element:

Table 8-46 – FlowElementsContainer model associations

Attribute Name	Description/Usage
flowElements: FlowElement [0..*]	This association specifies the particular *flow elements* contained in a FlowElementContainer. *Flow elements* are Events, Gateways, Sequence Flow, Activities, Data Objects, Data Associations, and Choreography Activities. Note that: • Choreography Activities MUST NOT be included as a flowElement for a Process • Activities, Data Associations, and Data Objects MUST NOT be included as a flowElement for a Choreography.
artifacts: Artifact [0..*]	The list of Artifacts that are contained within the FlowElementsContainer (which could be a Process or Choreography)

8.3.10. Gateways

Gateways are used to control how the Process flows (how *Tokens* flow) through Sequence Flow as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term “gateway” implies that there is a gating mechanism that either allows or disallows passage through the Gateway--that is, as *tokens* arrive at a Gateway, they can be merged together on input and/or split apart on output as the Gateway mechanisms are invoked.

Gateways, like Activities, are capable of consuming or generating additional control *tokens*, effectively controlling the execution semantics of a given Process. The main difference is that Gateways do not represent ‘work’ being done and they are considered to have zero effect on the operational measures of the Process being executed (cost, time, etc.).

The Gateway controls the flow of both diverging and converging Sequence Flow. That is, a single Gateway could have multiple input and multiple output flows. Modelers and modeling tools may want to enforce a best practice of a Gateway only performing one of these functions. Thus, it would take two sequential Gateways to first converge and then to diverge the Sequence Flow.

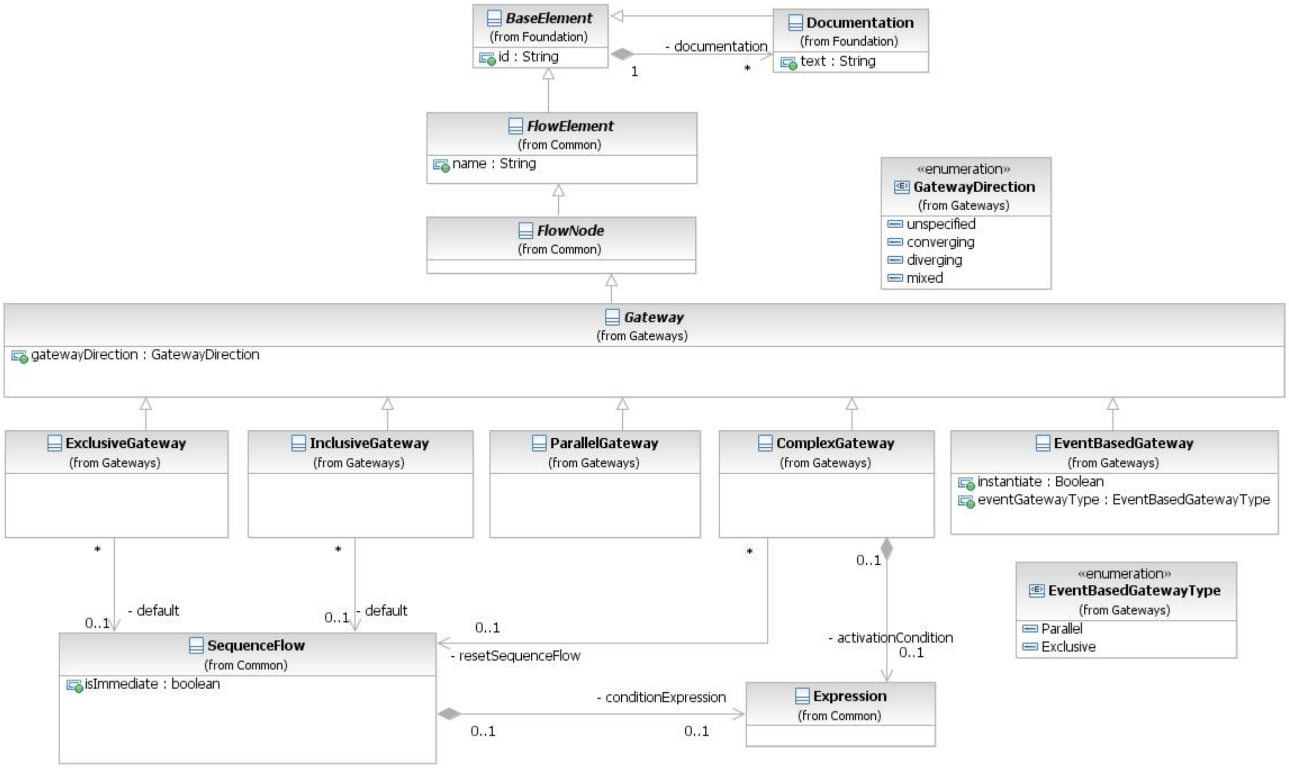


Figure 8-25 – Gateway class diagram

The details for the types of Gateways (Exclusive, Inclusive, Parallel, Event-Based, and Complex) is defined on page 295 for Processes and on page 375 for Choreographies.

The Gateway class is an abstract type. Its concrete subclasses define the specific semantics of individual Gateway types, defining how the Gateway behaves in different situations.

The Gateway element inherits the attributes and model associations of FlowElement (see Table 8-45). Table 8-47 presents the additional attributes of the Gateway element:

Table 8-47 – Gateway attributes

Attribute Name	Description/Usage
gatewayDirection: GatewayDirection = unspecified { unspecified | converging | diverging | mixed }	An attribute that adds constraints on how the gateway may be used. • **unspecified**: There are no constraints. The Gateway MAY have any number of *incoming* and *outgoing* Sequence Flow. • **converging**: This Gateway MAY have multiple *incoming* Sequence Flow but MUST have no more than one *outgoing* SequenceFlow. • **diverging**: This Gateway MAY have multiple outgoing Sequence Flow but MUST have no more than one *incoming* Sequence Flow. • **mixed**: This Gateway contains multiple *outgoing* and multiple *incoming* Sequence Flow.

8.3.11. Interaction Specification

The `InteractionSpecification` element is a superclass for all elements that require definition about interactions between *Participants*. Specifically, the `InteractionSpecification` element defines a list of *Participants* and Message Flow. The elements that inherit from `InteractionSpecification` include Collaboration, Choreography, and Conversation.

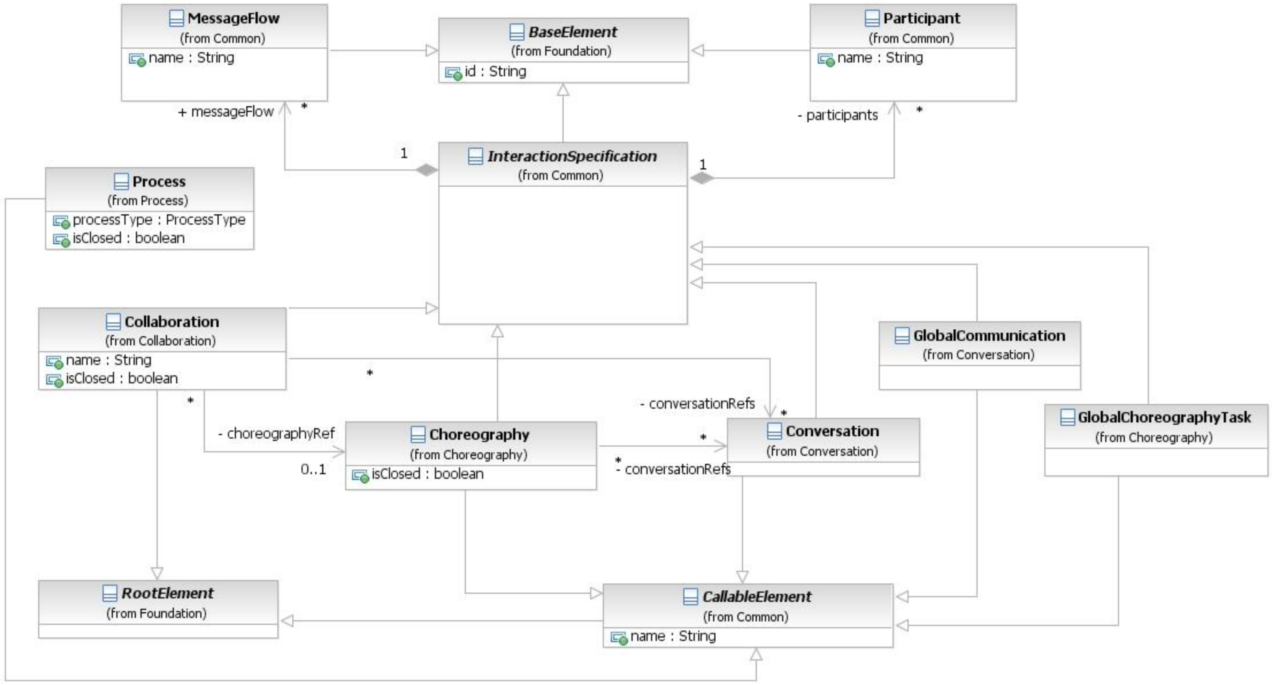


Figure 8-26 – InteractionSpecification class diagram

The `InteractionSpecification` element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 8-48 presents the attributes and model associations of the `InteractionSpecification` element:

Table 8-48 – InteractionSpecification attributes and model associations

Attribute Name	Description/Usage
participants: Participant [0..*]	This provides the list of *Participants* that are used in the `InteractionSpecification`
messageFlow: Message Flow [0..*]	This provides the list of Message Flow that are used in the `InteractionSpecification`

8.3.12. Item Definition

BPMN elements, such as DataObjects and Messages, represent items that are manipulated, transferred, transformed or stored during Process flows. These items can be either physical items, such as the mechanical part of a vehicle, or information items such the catalog of the mechanical parts of a vehicle.

An important characteristics of items in Process is their structure. BPMN does not require a particular format for this data structure, but it does designate XML Schema as its default. The structure attribute references the actual data structure.

The default format of the data structure for all elements can be specified in the Definitions element using the typeLanguage attribute. For example, a typeLanguage value of *http://www.w3.org/2001/XMLSchema*" indicates that the data structures using by elements within that Definitions are in the form of XML Schema types. If unspecified, the default is XML schema. An Import is used to further identify the location of the data structure (if applicable). For example, in the case of data structures contributed by an XML schema, an Import would be used to specify the file location of that schema.

Structure definitions are always defined as separate entities, so they cannot be inlined in one of their usages. You will see that in every mention of structure definition there is a "reference" to the element. This is why this class inherits from RootElement.

An ItemDefinition element can specify an import reference where the proper definition of the structure is defined.

In cases where the data structure represents a collection, the multiplicity can be projected into the attribute isCollection. If this attribute is set to "*true*", but the actual type is not a collection type, the model is considered as invalid. BPMN compliant tools might support an automatic check for these inconsistencies and report this as an error. The default value for this element is "*false*".

The itemKind attribute specifies the nature of an item which can be a physical or an information item.

Figure 8-27 shows the ItemDefinition class diagram. When a ItemDefinition is defined it is contained in Definitions.

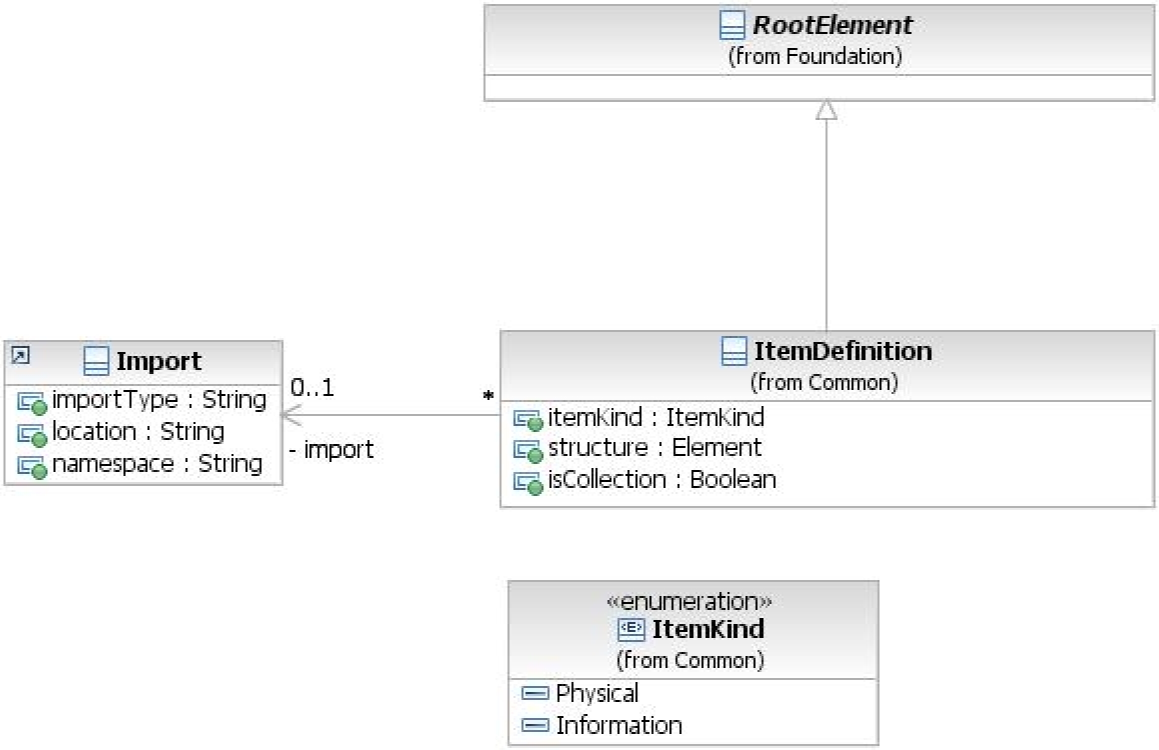


Figure 8-27 – ItemDefinition class diagram

The `ItemDefinition` element inherits the attributes and model associations `BaseElement` (see Table 8-5) through its relationship to `RootElement`. Table 8-49 presents the additional attributes and model associations for the `ItemDefinition` element:

Table 8-49 – ItemDefinition attributes & model associations

Attribute Name	Description/Usage
itemKind: ItemKind = “Information” {Information | Physical}	This defines the nature of the Item. Possible values are `Physical` or `Information`. The default value is `Information`.
structure: Element [0..1]	The concrete data structure to be used.
import: Import [0..1]	Identifies the location of the data structure and its format. If the `importType` attribute is left unspecified, the `typeLanguage` specified in the `Definitions` that contains this `ItemDefinition` is assumed
isCollection: boolean = False	Setting this flag to *true* indicates that the actual data type is a collection.

8.3.13. Message

A Message represents the content of a communication between two *Participants*. In BPMN 2.0, a Message is a graphical object (it was a supporting element in BPMN 1.2). An ItemDefinition is used to specify the Message structure.

When displayed in a diagram:

- In a Message is a rectangle with converging diagonal lines in the upper half of the rectangle to give the appearance of an envelope (see Figure 8-28). It MUST be drawn with a single thin line.
  - The use of text, color, size, and lines for a Task MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

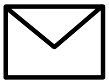


Figure 8-28 – A Message

In addition, when used in a Choreography Diagram more than one Message may be used for a single Choreography Task or a Choreography Sub-Process. In this case, it is important to know the first (initiating) Message of the interaction. For return (non-initiating) Messages the symbol of the Message is shaded with a light fill (see Figure 8-29).

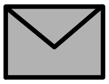


Figure 8-29 – An non-*initiating* Message

- Any Message that is after first on the list of Messages for a Choreography Task or a Choreography Sub-Process MUST be shaded with a light fill.

In a Collaboration, the communication itself is represented by a Message Flow (see the Section "Message Flow" below for more details). The Message can be displayed as attached (Associated) to a Message Flow in a Collaboration (see Figure 8-30 and Figure 8-31).

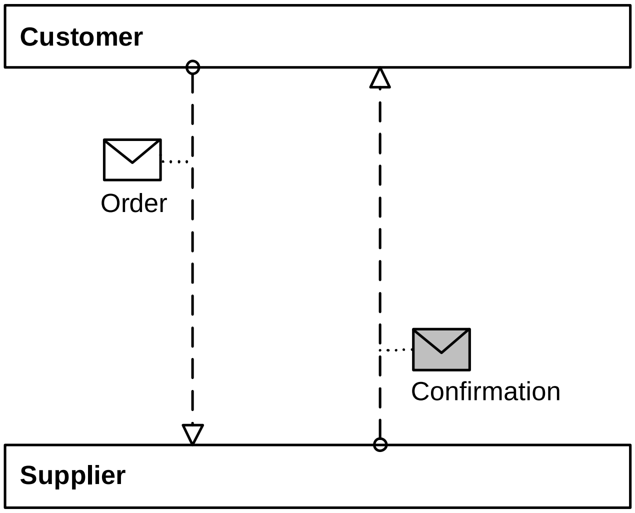


Figure 8-30 –Messages shown Associated with Message Flow

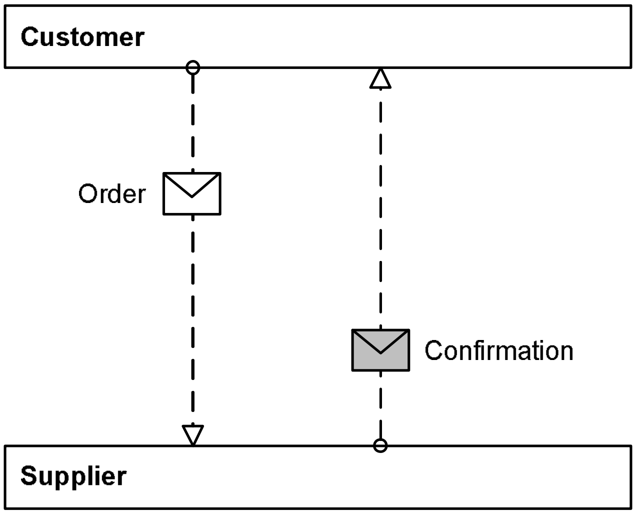


Figure 8-31 –Messages Association overlapping Message Flow

In a Choreography, the communication is represented by a Choreography Task (see page 350). The Message can be displayed as Associated with a Choreography Task in a Choreography (see

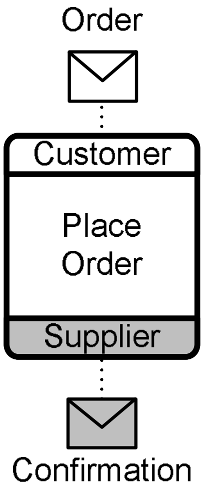


Figure 8-32 –Messages shown Associated with a Choreography Task

In a Process that is not used in a Collaboration, the communication is not displayed, but a Message can be defined for Activities that send and receive Messages (such as a Send Task—see Figure 8-33). Note that the display of Messages in a Process, Collaboration, or Choreography is optional.

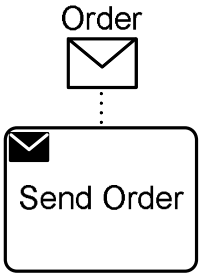


Figure 8-33 –Messages shown Associated with a Send Task

Figure 8-34 displays the class diagram showing the attributes and model associations for the Message element.

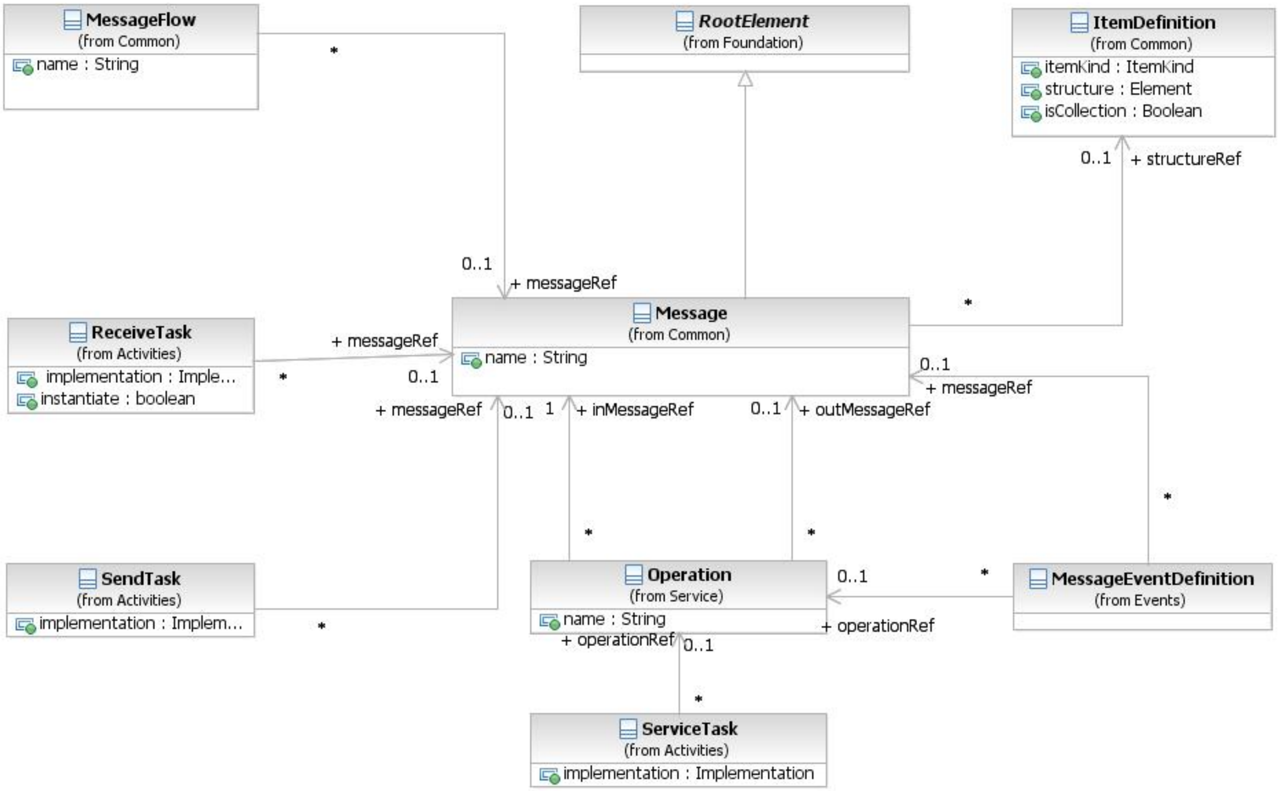


Figure 8-34 – The Message class diagram

The Message element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-50 presents the additional attributes and model associations for the Message element:

Table 8-50 – Message attributes and model associations

Attribute Name	Description/Usage
name: string	Name is a text description of the Message
structureRef : ItemDefinition [0..1]	An ItemDefinition is used to define the “payload” of the Message.

8.3.14. Message Flow

A Message Flow is used to show the flow of Messages between two Participants that are prepared to send and receive them.

- A Message Flow MUST connect two separate Pools. They connect either to the Pool boundary or to Flow Objects within the Pool boundary. They MUST NOT connect two objects within the same Pool.
- A Message Flow is a line with an open circle line start and an open arrowhead line end that MUST be drawn with a dashed single line (see Figure 8-35).
  - The use of text, color, size, and lines for a Pool MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

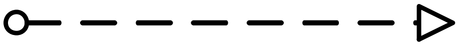


Figure 8-35 – A Message Flow

In Collaboration Diagrams (the view showing the Choreography Process Combined with Orchestration Processes), the Message Flow can be extended to show the Message that is passed from one Participant to another (see Figure 8-36).

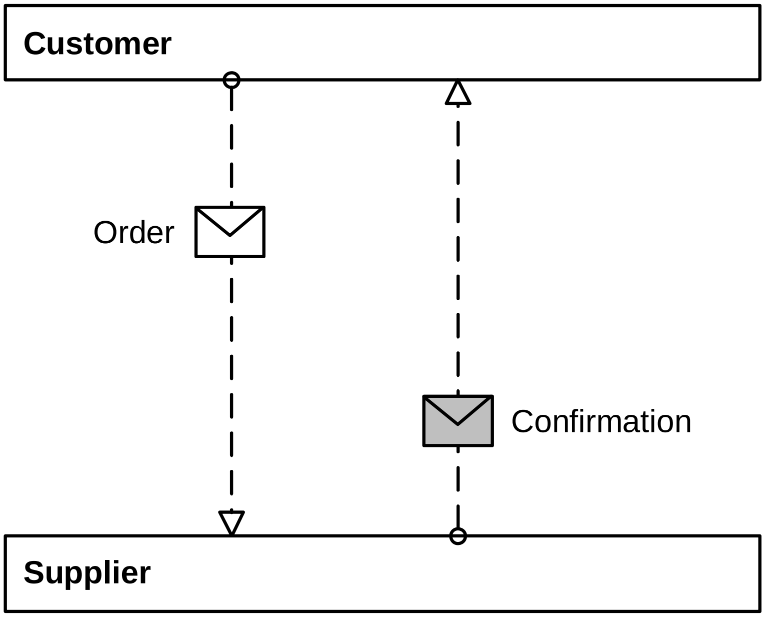


Figure 8-36 – A Message Flow with an Attached Message

If a Choreography is included in the Collaboration, then the Message Flow will "pass-through" a Choreography Task as it connects from one Participant to another (see Figure 8-37).

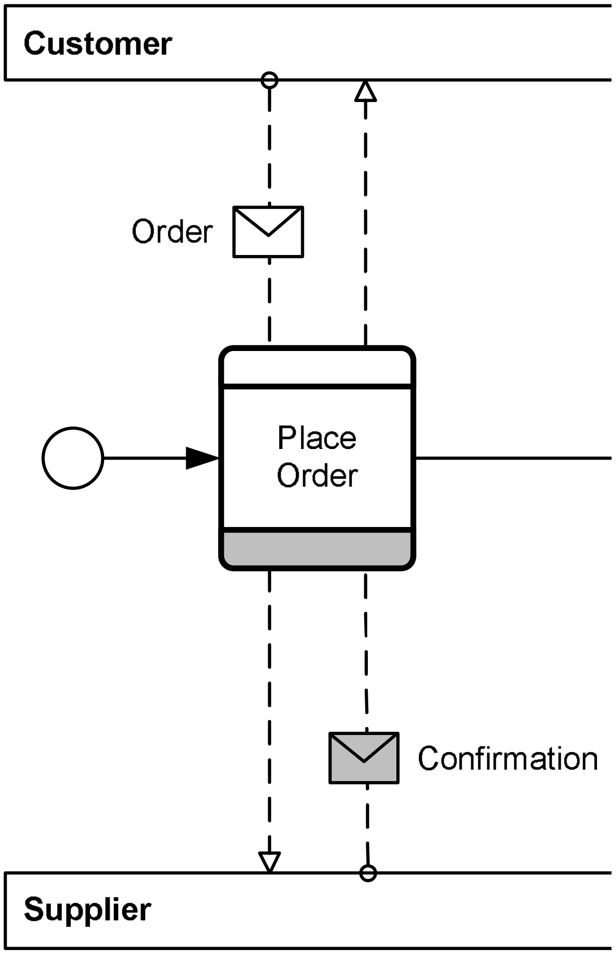


Figure 8-37 – A Message Flow passing through a Choreography Task

Figure 8-38 displays the class diagram of Message Flow and its relationships to other BPMN elements. When a Message Flow is defined it is contained either within a Collaboration, a Choreography Task, or a GlobalChoreographyTask.

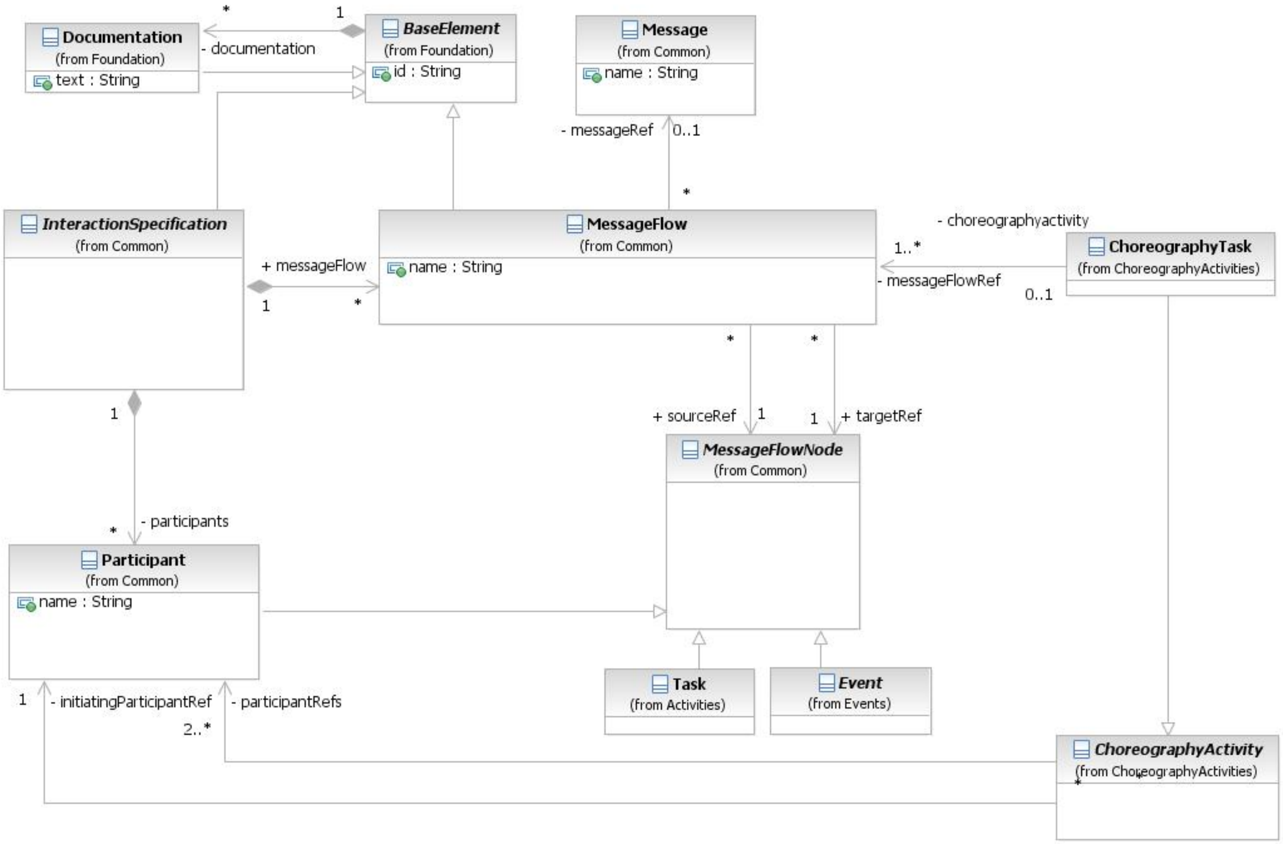


Figure 8-38 – The Message Flow Class Diagram

The Message Flow element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-51 presents the additional attributes and model associations for the Message Flow element:

Table 8-51 – Message Flow attributes and model associations

Attribute Name	Description/Usage
name: string	Name is a text description of the Message Flow.
sourceRef: MessageFlowNode	The MessageFlowNode that the Message Flow is connecting from. Of the types of MessageFlowNode, only Pools/*Participants*, Activities, and Events can be the *source*.
targetRef: MessageFlowNode	The MessageFlowNode that the Message Flow is connecting to. Of the types of MessageFlowNode, only Pools/*Participants*, Activities, and Events can be the *target*.
messageRef: Message [0..1]	The messageRef model association defines the Message that is passed via the Message Flow. See page 112 for more details.

Message Flow Node

The MessageFlowNode element is used to provide a single element as the source and target Message Flow associations (see Figure 8-38, above) instead of the individual associations of the elements that can connect to Message Flow (see the section above). Only the Pool/*Participant*, Activity, and Event elements can connect to Message Flow and thus, these elements are the only ones that are sub-classes of MessageFlowNode.

The MessageFlowNode element does not have any attributes or model associations and does not inherit from any other BPMN element. Since Pools/*Participants*, Activities, and Events have their own attributes, model associations, and inheritances, additional attributes and model associations for the MessageFlowNode element are not necessary.

Message Flow Associations

These elements are used to do mapping between two elements that both contain Message Flow. The MessageFlowAssociation provides the mechanism to match up the Message Flow.

A MessageFlowAssociation is used when an (*outer*) diagram with Message Flow contains an (*inner*) diagram that also has Message Flow. There are three (3) usages of MessageFlowAssociation. It is used when:

- A Collaboration references a Choreography for inclusion between the Collaboration's Pools (*Participants*). The Message Flow of the Choreography (the inner diagram) need to be mapped to the Message Flow of the Collaboration (the outer diagram).
- A Collaboration references a Conversation that contains Message Flow. The Message Flow of the Conversation may serve as a partial requirement for the Collaboration. Thus, the Message Flow of the Conversation (the inner diagram) need to be mapped to the Message Flow of the Collaboration (the outer diagram).
- A Choreography references a Conversation that contains Message Flow. The Message Flow of the Conversation may serve as a partial requirement for the Choreography. Thus, the Message Flow of the Conversation (the inner diagram) need to be mapped to the Message Flow of the Choreography (the outer diagram).

Figure 8-39 shows the class diagram for the MessageFlowAssociation element.

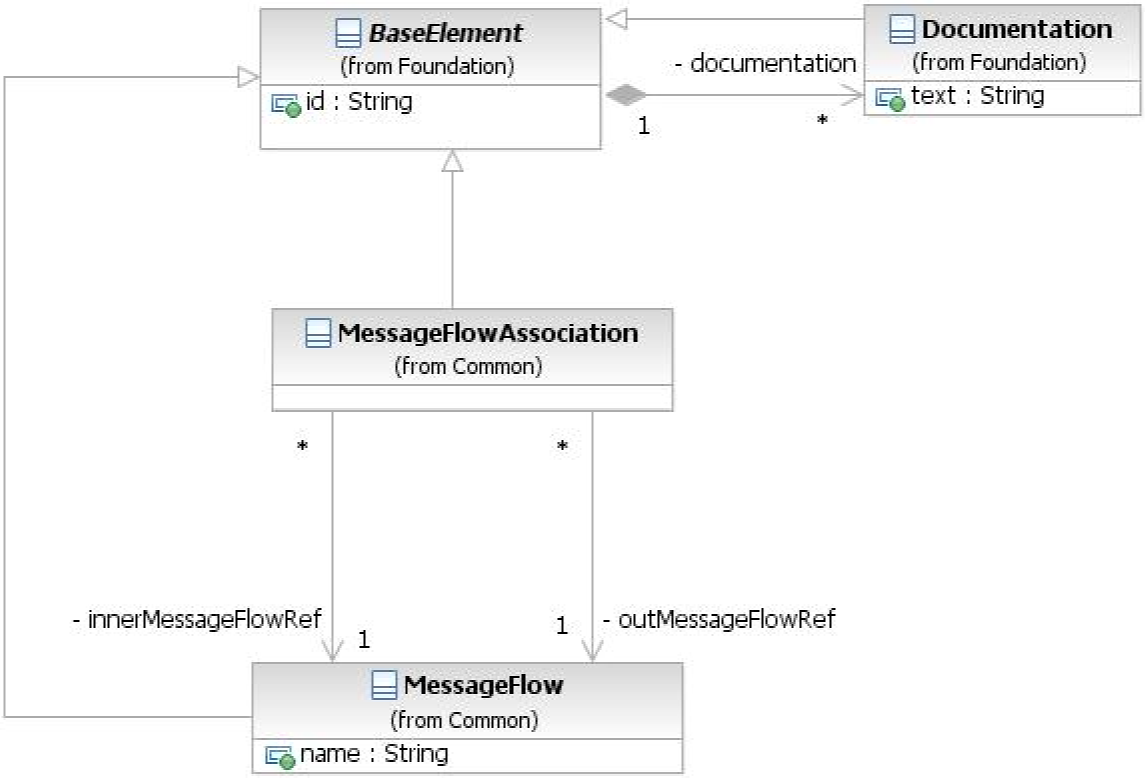


Figure 8-39 – MessageFlowAssociation class diagram

The MessageFlowAssociation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-53 presents the additional model associations for the MessageFlowAssociation element:

Table 8-52 – MessageFlowAssociation attributes and model associations

Attribute Name	Description/Usage
innerMessageFlowRef: Message Flow	This attribute defines the Message Flow of the referenced element (e.g., a Choreography to be used in a Collaboration) that will be mapped to the parent element (e.g., the Collaboration).
outerMessageFlowRef: Message Flow	This attribute defines the Message Flow of the parent element (e.g., a Collaboration references a Choreography) that will be mapped to the referenced element (e.g., the Choreography).

8.3.15. Participants

A *Participant* represents a specific `PartnerEntity` (e.g., a company) and/or a more general `PartnerRole` (e.g., a buyer, seller, or manufacturer) that *Participants* in a Collaboration. A *Participant* is often responsible for the execution of the Process enclosed in a Pool; however, a Pool may be defined without a Process.

Figure 8-40 displays the class diagram of the *Participant* and its relationships to other BPMN elements. When *Participants* are defined they are contained within an `InteractionSpecification`, which includes the sub-types of Collaboration, a Choreography, a Conversation, a `Global Communication`, or a `GlobalChoreographyTask`.

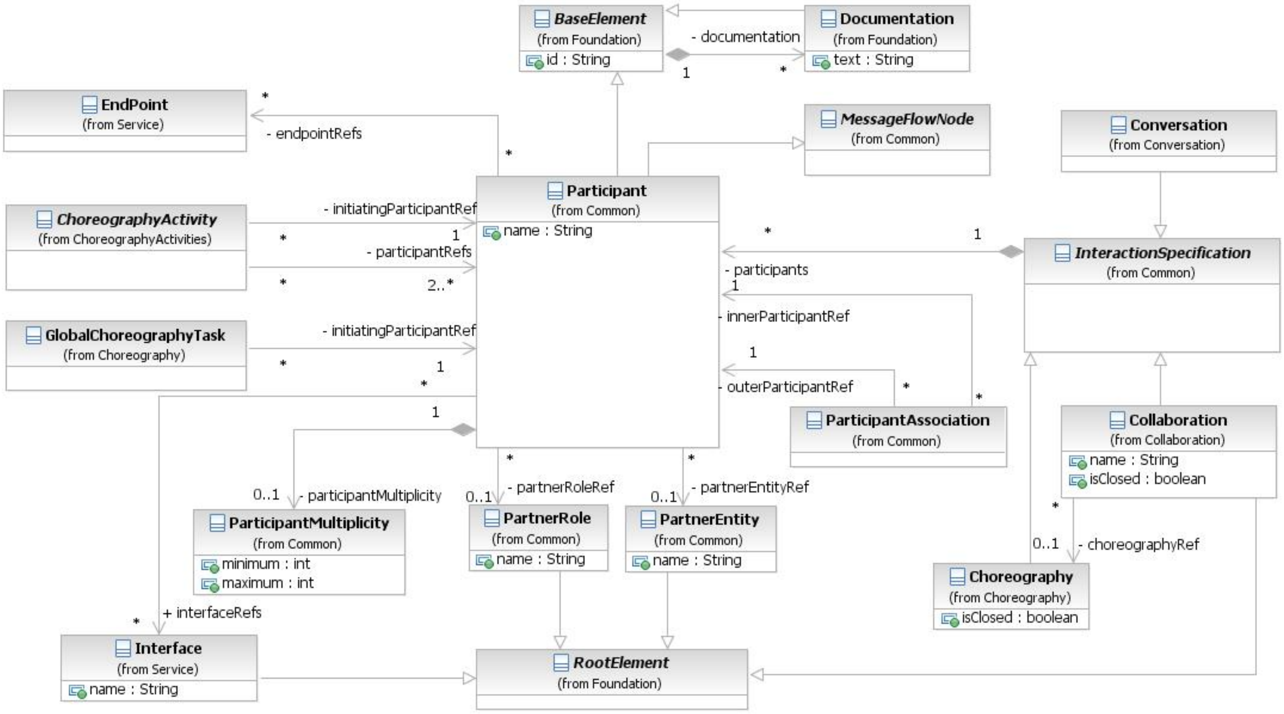


Figure 8-40 – The Participant Class Diagram

The *Participant* element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 8-53 presents the additional attributes and model associations for the *Participant* element:

Table 8-53 – Participant attributes and model associations

Attribute Name	Description/Usage
name: string [0..1]	`Name` is a text description of the *Participant*. The `name` of the *Participant* can be displayed directly or it can be substituted by the associated `PartnerRole` or `PartnerEntity`. Potentially, both the `PartnerEntity name` and `PartnerRole name` can be displayed for the *Participant*.
processRef: Process [0..1]	The `processRef` attribute identifies the `Process` that the `Participant` uses in the *Collaboration*. The `Process` will be displayed within the *Participant's* `Pool`.
partnerRoleRef: PartnerRole [0..1]	The `partnerRoleRef` attribute identifies a `PartnerRole` that the *Participant* plays in the Collaboration. Both a `PartnerRole` and a `PartnerEntity` MAY be defined for the *Participant*.
partnerEntityRef: PartnerEntity [0..1]	The `partnerEntityRef` attribute identifies a `PartnerEntity` that the *Participant* plays in the *Collaboration*. Both a `PartnerRole` and a `PartnerEntity` MAY be defined for the *Participant*.
interfaceRef: Interface [0..*]	This association defines `Interfaces` that a *Participant* supports. The definition of `Interfaces` is provided on page 140.
participantMultiplicity: participantMultiplicity [0..1]	The `participantMultiplicityRef` model association is used to define *Participants* that represent more than one (1) instance of the *Participant* for a given interaction. See the next section for more details on `ParticipantMultiplicity`.
endpointRefs: EndPoint [0..*]	This attribute is used to specify the address (or endpoint reference) of concrete services realizing the *Participant*.

PartnerEntity

An `PartnerEntity` is one of the possible types of *Participant* (see the section above).

The PartnerEntity element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-54 presents the additional attributes and model associations for the PartnerEntity element:

Table 8-54 – PartnerEntity attributes

Attribute Name	Description/Usage
name: string	Name is a text description of the PartnerEntity.

PartnerRole

A PartnerRole is one of the possible types of *Participant* (see the section above).

The PartnerRole element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-55 presents the additional attributes and model associations for the PartnerRole element:

Table 8-55 – PartnerRole attributes

Attribute Name	Description/Usage
name: string	Name is a text description of the PartnerRole.

Participant Multiplicity

ParticipantMultiplicity is used to define the multiplicity of a Participant.

For example, a manufacturer may request a quote from multiple suppliers in a Choreography.

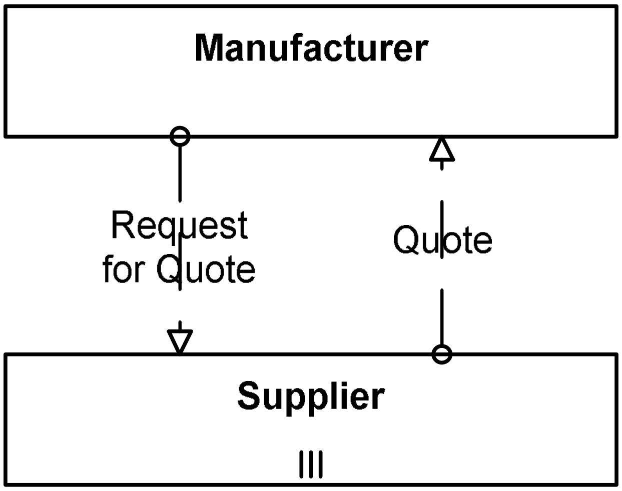


Figure 8-41 – A Pool with a Multiple Participant

The following figure shows the *Participant* class diagram.

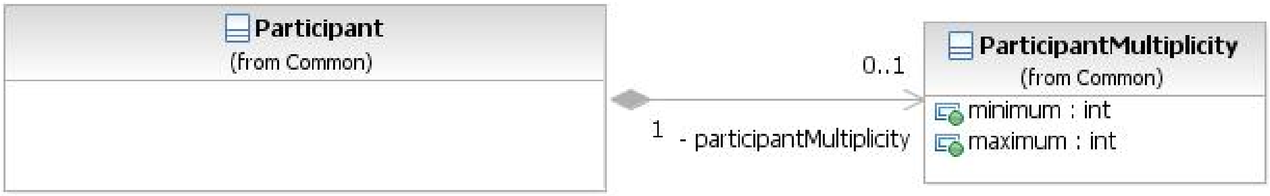


Figure 8-42 – The Participant Multiplicity class diagram

When the minimum attribute of the `ParticipantMultiplicty` element has been set by the modeler, then the *multi-instance* marker will be displayed in bottom center of the `Pool` (*Participant* – see Figure 8-41) or the `Participant Band` of a `Choreography Activity` (see page 356).

Table 8-56 presents the attributes for the `ParticipantMultiplicity` element:

Table 8-56 – ParticipantMultiplicity attributes

Attribute Name	Description/Usage
minimum: integer [0..1] = 2	The `minimum` attribute defines minimum number of `Participants` that MUST be involved in the `Collaboration`. The value of `minimum` MUST be two (2) or greater.
maximum: integer [0..1] = 2	The `maximum` attribute defines maximum number of `Participants` that MAY be involved in the `Collaboration`. The value of `maximum` MUST be two (2) or greater.

Table 8-57 presents the *Instance* attributes of the `ParticipantMultiplicity` element:

Table 8-57 – ParticipantMultiplicity *Instance* attributes

Attribute Name	Description/Usage
numParticipants: integer [0..1]	The current number of the multiplicity of the *Participant* for this `Choreography` or `Collaboration` *Instance*.

ParticipantAssociation

These elements are used to do mapping between two elements that both contain *Participants*. There are situations where the *Participants* in different diagrams may be defined differently because they were developed independently, but represent the same thing. The `ParticipantAssociation` provides the mechanism to match up the *Participants*.

A `ParticipantAssociation` is used when an (*outer*) diagram with *Participants* contains an (*inner*) diagram that also has *Participants*. There are three (3) usages of `ParticipantAssociation`. It is used when:

- A Collaboration references a Choreography for inclusion between the Collaboration's Pools (*Participants*). The *Participants* of the Choreography (the inner diagram) need to be mapped to the *Participants* of the Collaboration (the outer diagram).
- A Collaboration references a Process (within one of its Pools) and that Process contains a Call Activity that references another Process that has a *definitional* Collaboration. The *Participants* of the *definitional* Collaboration for the called Process (the inner diagram) need to be mapped to the *Participants* of the Collaboration (the outer diagram). Note that the outer Collaboration may be a *definitional* Collaboration for the referenced Process.
- A Choreography contains a Call Choreography Activity that references another Choreography. The *Participants* of the called Choreography (the inner diagram) need to be mapped to the *Participants* of the calling Choreography (the outer diagram).

A `ParticipantAssociation` can be owned by the outer diagram or one its elements. Figure 8-43 shows the class diagram for the `ParticipantAssociation` element.

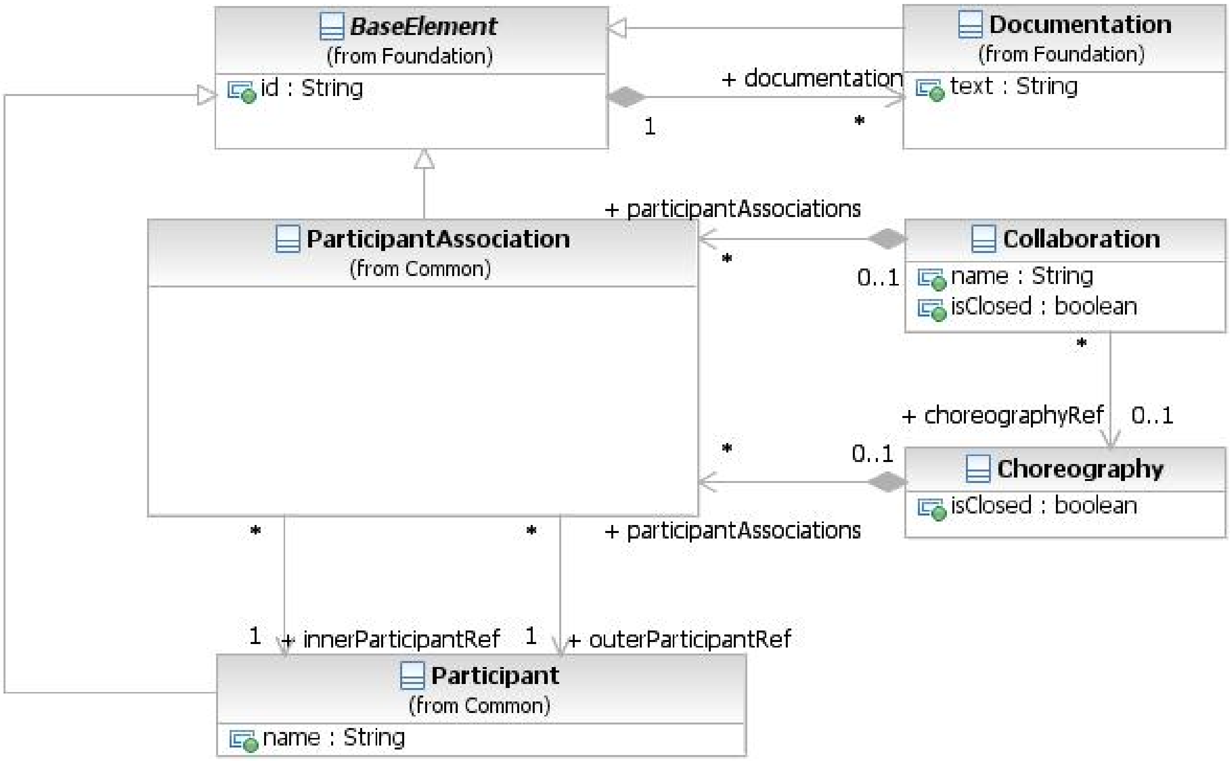


Figure 8-43 – ParticipantAssociation class diagram

The ParticipantAssociation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-58 presents the additional model associations for the ParticipantAssociation element:

Table 8-58 – ParticipantAssociation model associations

Attribute Name	Description/Usage
innerParticipantRef: Participant	This attribute defines the *Participant* of the referenced element (e.g., a Choreography to be used in a Collaboration) that will be mapped to the parent element (e.g., the Collaboration).
outerParticipantRef: Participant	This attribute defines the *Participant* of the parent element (e.g., a Collaboration references a Choreography) that will be mapped to the referenced element (e.g., the Choreography).

8.3.16. Resources

The Resource class is used to specify resources that can be referenced by Activities. These Resources can be Human Resources as well as any other resource assigned to Activities during Process execution time.

The definition of a Resource is "abstract", because it only defines the Resource, without detailing how e.g. actual user IDs are associated at runtime. Multiple Activities can utilize the same Resource.

Every Resource can define a set of ResourceParameters. These parameters can be used at runtime to define query e.g. into an Organizational Directory. Every Activity referencing a parameterized Resource can bind values available in the scope of the Activity to these parameters.

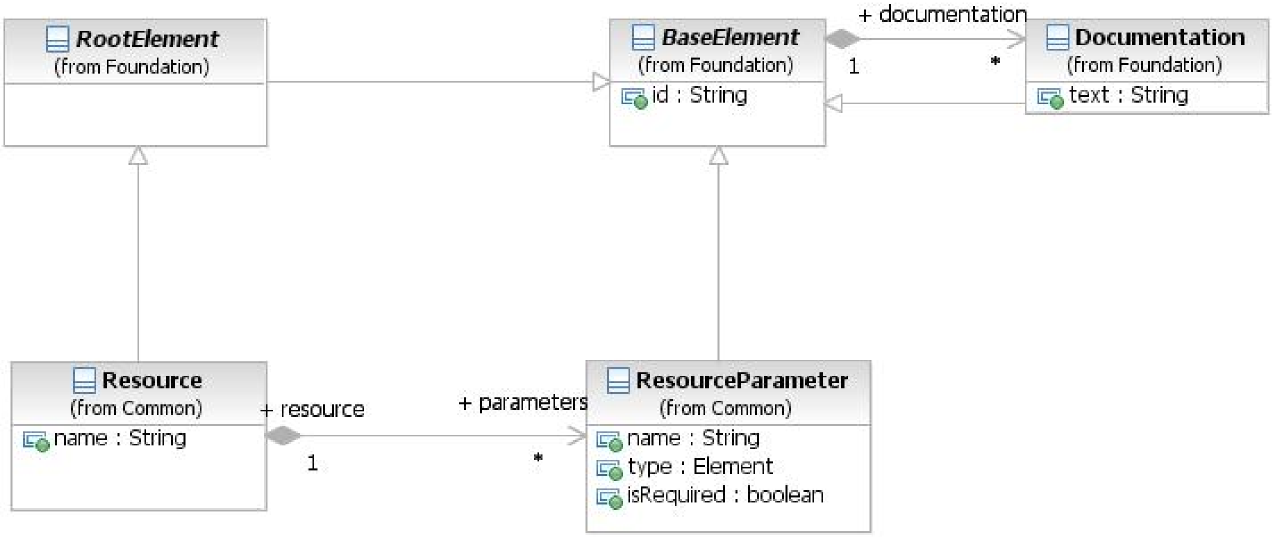


Figure 8-44 – Resource class diagram

The Resource element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-59 presents the additional model associations for the Resource element:

Table 8-59 – Resource attributes and model associations

Attribute Name	Description/Usage
name: string	This attribute specifies the name of the Resource.
parameters: ResourceParameter [0..*]	This model association specifies the definition of the parameters required at runtime to resolve the Resource.

As mentioned before, the Resource can define a set of parameters to define a query to resolve the actual resources (e.g. user ids).

The ResourceParameter element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement. Table 8-59 presents the additional model associations for the ResourceParameter element:

Table 8-60 – ResourceParameter attributes and model associations

Attribute Name	Description/Usage
name: String	Specifies the name of the query parameter.
type: Element	Specifies the type of the query parameter.
isRequired: Boolean	Specifies, if a parameter is optional or mandatory.

8.3.17. Sequence Flow

A Sequence Flow is used to show the order of Flow Elements in a Process or a Choreography. Each Sequence Flow has only one *source* and only one *target*. The *source* and *target* must be from the set of the following Flow Elements: Events (Start, Intermediate, and End), Activities (Task and Sub-Process; for Processes), Choreography Activities (Choreography Task and Choreography Sub-Process; for Choreographies), and Gateways.

- A Sequence Flow is line with a solid arrowhead that MUST be drawn with a solid single line (as seen in Figure 8-45).
  - The use of text, color, size, and lines for a Task MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63.

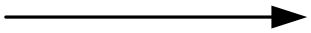


Figure 8-45 – A Sequence Flow

A Sequence Flow can optionally define a condition Expression, indicating that the *token* will be passed down the Sequence Flow only if the Expression evaluates to *true*. This Expression is typically used when the source of the Sequence Flow is a Gateway or an Activity.

- A *conditional outgoing* Sequence Flow from an Activity MUST be drawn with a mini-diamond marker at the beginning of the connector (as seen in Figure 8-46).
    - If a *conditional* Sequence Flow is used from a source Activity, then there MUST be at least one other *outgoing* Sequence Flow from that Activity.
- *Conditional outgoing* Sequence Flow from a Gateway MUST NOT be drawn with a mini-diamond marker at the beginning of the connector.
    - A source Gateway MUST NOT be of type Parallel or Event.

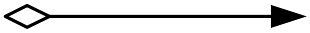


Figure 8-46 – A Conditional Sequence Flow

A Sequence Flow that has an Exclusive, Inclusive, or Complex Gateway or an Activity as its source can also be defined with as *default*. Such Sequence Flow will have a marker to show that it is a *default* flow. The *default* Sequence Flow is taken (a token is passed) only if all the other outgoing Sequence Flow from the Activity or Gateway are not valid (i.e., their condition Expressions are *false*)

- A *default outgoing* Sequence Flow MUST be drawn with a slash marker at the beginning of the connector (as seen in Figure 8-47).

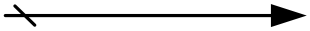


Figure 8-47 – A Default Sequence Flow

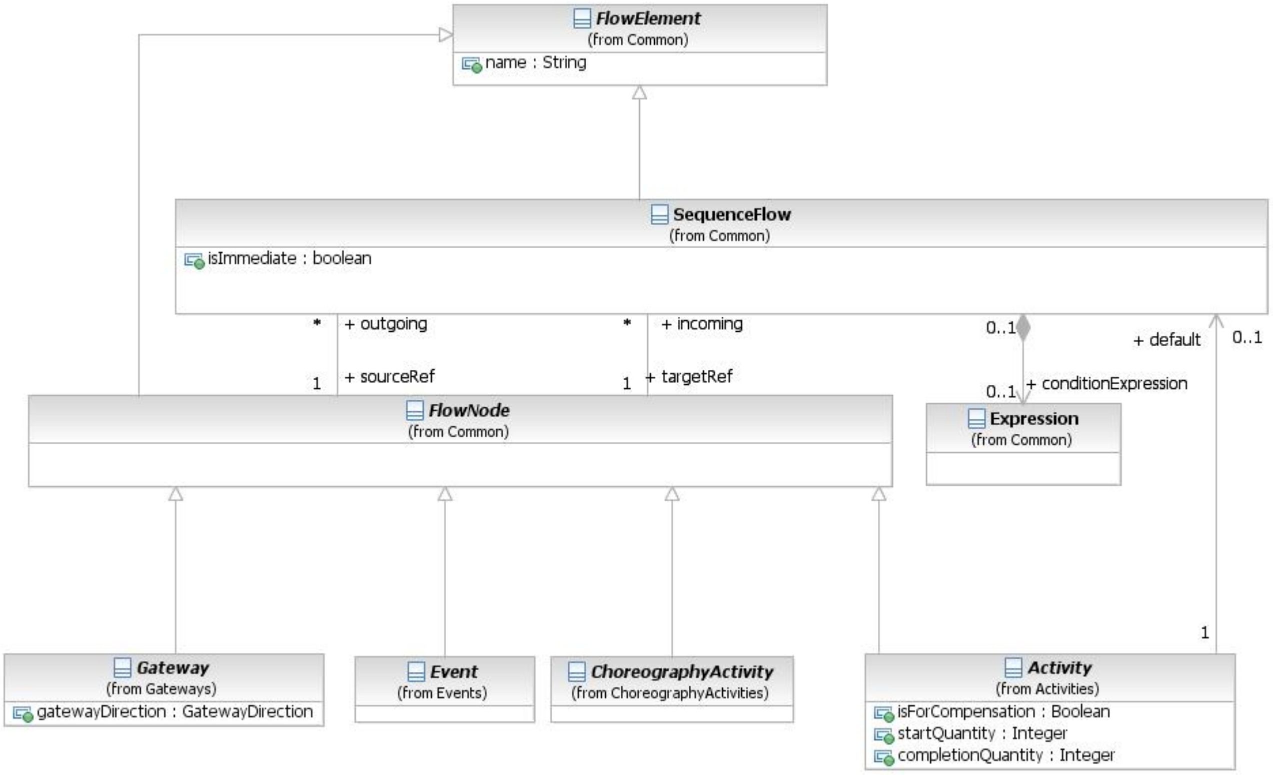


Figure 8-48 – SequenceFlow class diagram

The Sequence Flow element inherits the attributes and model associations of FlowElement (see Table 8-45). Table 8-61 presents the additional attributes and model associations of Sequence Flow element:

Table 8-61 – SequenceFlow attributes and model associations

Attribute Name	Description/Usage
sourceRef: FlowNode	The FlowNode that the Sequence Flow is connecting from. For a Process: Of the types of FlowNode, only Activities, Gateways, and Events can be the *source*. However, Activities that are Event Sub-Processes are not allowed to be a *source*. For a Choreography: Of the types of FlowNode, only Choreography Activities, Gateways, and Events can be the *source*.
targetRef: FlowNode	The FlowNode that the Sequence Flow is connecting to. For a Process: Of the types of FlowNode, only Activities, Gateways, and Events can be the *target*. However, Activities that are Event Sub-Processes are not allowed to be a *target*. For a Choreography: Of the types of FlowNode, only Choreography Activities, Gateways, and Events can be the *target*.

conditionExpression: Expression [0..1]	An optional Boolean Expression that acts as a gating *condition*. A *token* will only be placed on this Sequence Flow if this conditionExpression evaluates to *true*.
isImmediate: boolean [0..1]	An optional Boolean value specifying whether Activities or Choreography Activities not in the model containing the Sequence Flow can occur between the elements connected by the Sequence Flow. If the value is *true*, they MAY NOT occur. If the value is *false*, they MAY occur. Also see the isClosed attribute on Process, Choreography, and Collaboration. When the attribute has no value, the default semantics depends on the kind of model containing Sequence Flow: For a *public* Processes and Choreographies no value has the same semantics as if the value were *false*. For an *executable* and *non-executable* (internal) Processes no value has the same semantics as if the value were *true*. For *executable* Processes, the attribute MUST NOT be *false*.

Flow Node

The FlowNode element is used to provide a single element as the source and target Sequence Flow associations (see Figure 8-48, above) instead of the individual associations of the elements that can connect to Sequence Flow (see the section above). Only the Gateway, Activity, Choreography Activity, and Event elements can connect to Sequence Flow and thus, these elements are the only ones that are sub-classes of FlowNode.

Since Gateway, Activity, Choreography Activity, and Event have their own attributes, model associations, and inheritances; the FlowNode element does not inherit from any other BPMN element. Table 8-62 presents the additional model associations of the FlowNode element:

Table 8-62 – FlowNode model associations

Attribute Name	Description/Usage
incoming: Sequence Flow [0..*]	This attribute identifies the *incoming* Sequence Flow of the FlowNode.
outgoing: Sequence Flow [0..*]	This attribute identifies the *outgoing* Sequence Flow of the FlowNode.

8.3.18. Common Package XML Schemas

Table 8-63 – CallableElement XML schema

```
<xsd:element name="callableElement" type="tCallableElement"/>
<xsd:complexType name="tCallableElement">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element name="supportedInterfaceRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element ref="ioSpecification" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="ioBinding" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-64 – ConversationAssociation XML schema

```
<xsd:element name="conversationAssociation" type="tConversationAssociation"/>
<xsd:complexType name="tConversationAssociation">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="conversationRef" type="xsd:QName"/>
            <xsd:attribute name="correlationKeyRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-65 – Error XML schema

```
<xsd:element name="error" type="tError"/>
<xsd:complexType name="tError">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="structureRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-66 – Expression XML schema

```
<xsd:element name="expression" type="tExpression"/>
<xsd:complexType name="tExpression">
    <xsd:complexContent>
        <xsd:extension base="tBaseElementWithMixedContent"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-67 – FlowElement XML schema

```
<xsd:element name="flowElement" type="tFlowElement"/>
<xsd:complexType name="tFlowElement" abstract="true">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element ref="auditing" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="monitoring" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="categoryValue" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 8-68 – FlowNode XML schema

```
<xsd:element name="flowNode" type="tFlowNode"/>
<xsd:complexType name="tFlowNode" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tFlowElement">
            <xsd:sequence>
                <xsd:element name="incoming" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="outgoing" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-69 – FormalExpression XML schema

```
<xsd:element name="formalExpression" type="tFormalExpression" substitutionGroup="expression"/>
<xsd:complexType name="tFormalExpression" mixed="true">
    <xsd:complexContent>
        <xsd:extension base="tExpression">
            <xsd:attribute name="language" type="xsd:anyURI" use="optional"/>
            <xsd:attribute name="evaluatesToTypeRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-70 – InputOutputBinding XML schema

```
<xsd:element name="ioBinding" type="tinputOutputBinding"/>
<xsd:complexType name="tinputOutputBinding">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name=”inputDataRef" type="xsd:IDREF"/>
            <xsd:attribute name=”outputDataRef" type="xsd:IDREF"/>
            <xsd:attribute name=”operationRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-71 – ItemDefinition XML schema

```
<xsd:element name="itemDefinition" type="tItemDefinition" substitutionGroup="rootElement"/>
<xsd:complexType name="tItemDefinition">
      <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="structureRef" type="xsd:QName"/>
            <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
            <xsd:attribute name="itemKind" type="tItemKind" default="Information"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tItemKind">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Information"/>
        <xsd:enumeration value="Physical"/>
      </xsd:restriction>
</xsd:simpleType>
```

Table 8-72 – Message XML schema

```
<xsd:element name="message" type="tMessage" substitutionGroup="rootElement"/>
<xsd:complexType name="tMessage">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="structureRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-73 – MessageFlow XML schema

```
<xsd:element name="messageFlow" type="tMessageFlow"/>
<xsd:complexType name="tMessageFlow">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string" use="optional"/>
            <xsd:attribute name="sourceRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="targetRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="messageRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-74 – MessageFlowAssociation XML schema

```
<xsd:element name="messageFlowAssociation" type="tMessageFlowAssociation"/>
    <xsd:complexType name="tMessageFlowAssociation">
        <xsd:complexContent>
            <xsd:extension base="tBaseElement">
                <xsd:attribute name="innerMessageFlowRef" type="xsd:QName" use="required"/>
            <xsd:attribute name="outerMessageFlowRef" type="xsd:QName" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-75 – Participant XML schema

```
<xsd:element name="participant" type="tParticipant"/>
<xsd:complexType name="tParticipant">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="interfaceRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="endPointRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element ref="participantMultiplicity" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="partnerRoleRef" type="xsd:QName" use="optional"/>
            <xsd:attribute name="partnerEntityRef" type="xsd:QName" use="optional"/>
            <xsd:attribute name="processRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-76 – ParticipantAssociation XML schema

```
<xsd:element name="participantAssociation" type="tParticipantAssociation"/>
<xsd:complexType name="tParticipantAssociation">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="innerParticipantRef" type="xsd:QName"/>
                <xsd:element name="outerParticipantRef" type="xsd:QName"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-77 – PartnerEntity XML schema

```
<xsd:element name="partnerEntity" type="tPartnerEntity" substitutionGroup="rootElement"/>
<xsd:complexType name="tPartnerEntity">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-78 – PartnerRole XML schema

```
<xsd:element name="partnerRole" type="tPartnerRole" substitutionGroup="rootElement"/>
<xsd:complexType name="tPartnerRole">
      <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 8-79 – Resources XML schema

```
<xsd:element name="resource" type="tResource" substitutionGroup="rootElement"/>
<xsd:complexType name="tResource">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element ref="resourceParameter" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-80 – SequenceFlow XML schema

```
<xsd:element name="sequenceFlow" type="tSequenceFlow" substitutionGroup="flowElement"/>
<xsd:complexType name="tSequenceFlow">
        <xsd:complexContent>
          <xsd:extension base="tFlowElement">
               <xsd:sequence>
<xsd:element name="conditionExpression"  type="tExpression" minOccurs="0" maxOccurs="1"/>
               </xsd:sequence>
               <xsd:attribute name="sourceRef" type="xsd:IDREF" use="required"/>
               <xsd:attribute name="targetRef" type="xsd:IDREF" use="required"/>
               <xsd:attribute name="isImmediate" type="xsd:boolean" default="true"/>
          </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

8.4. Services

The Service package contains constructs necessary for modeling services, interfaces, and operations.

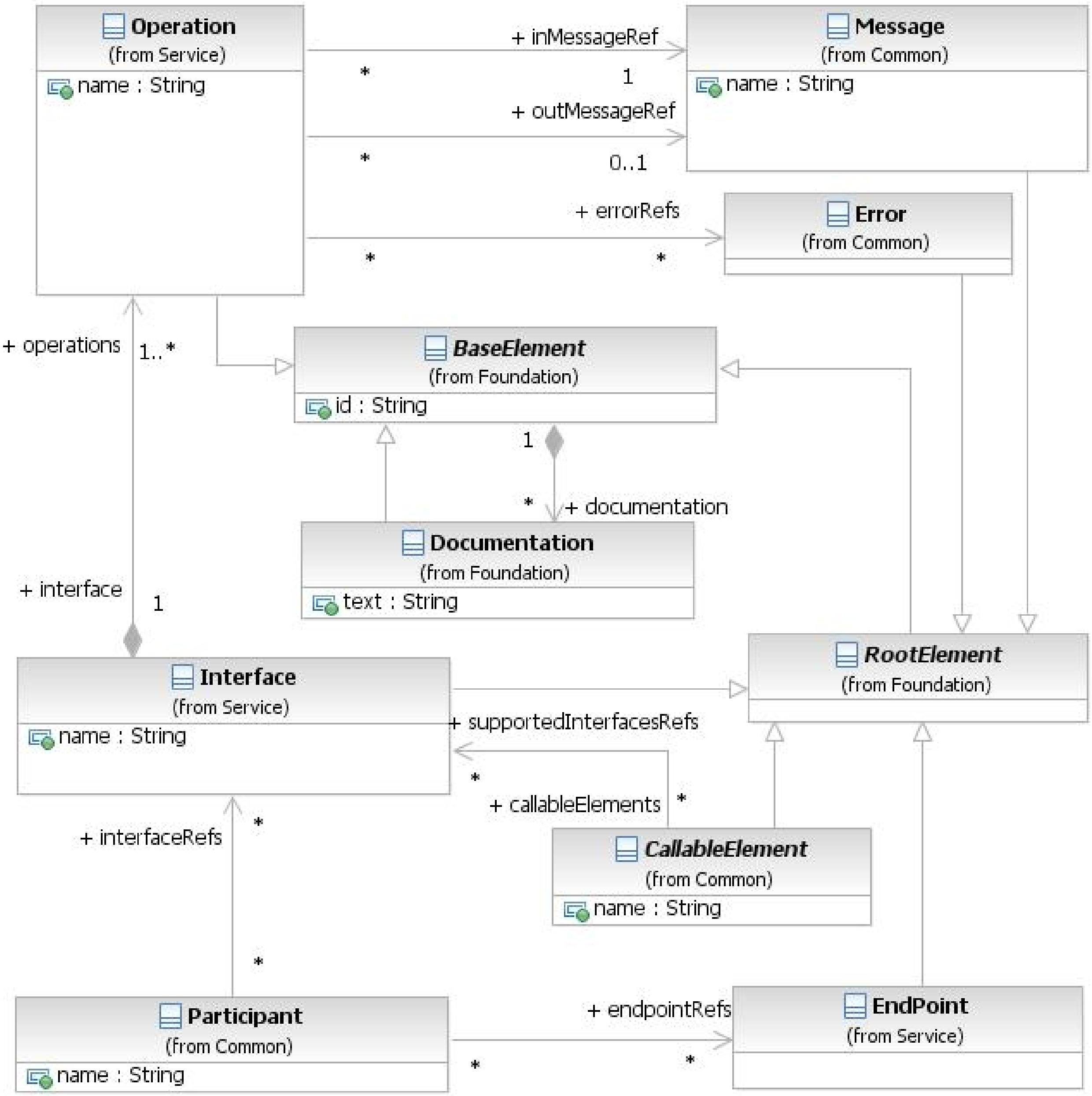


Figure 8-49 – The Service class diagram

8.4.1. Interface

An `Interface` defines a set of operations that are implemented by `Services`.

The `Interface` inherits the attributes and model associations of `BaseElement` (see Table 8-5) through its relationship to `RootElement`. Table 8-81 presents the additional attributes and model associations of the `Interface`:

Table 8-81 – Interface attributes and model associations

Attribute Name	Description/Usage
name: string	The descriptive name of the element.
operations: Operation [1..*]	This attribute specifies operations that are defined as part of the Interface. An Interface has at least one Operation.
callableElements: CallableElements [0..*]	The CallableElements that use this Interface.

8.4.2. EndPoint

The actual definition of the service address is out of scope of BPMN 2.0. The EndPoint element is an extension point and extends from RootElement. The EndPoint element may be extended with endpoint reference definitions introduced in other specifications (e.g. WS-Addressing).

EndPoints can be specified for *Participants*.

8.4.3. Operation

An Operation defines Messages that are consumed and, optionally, produced when the Operation is called. It may also define zero or more errors that are returned when operation fails. The Operation inherits the attributes and model associations of BaseElement (see Table 8-5). Table 8-82 below presents the additional attributes and model associations of the Operation:

Table 8-82 – Operation attributes and model associations

Attribute Name	Description/Usage
name: string	The descriptive name of the element.
inMessageRef: Message	This attribute specifies the input Message of the Operation. An Operation has exactly one input Message.
outMessageRef: Message [0..1]	This attribute specifies the output Message of the Operation. An Operation has at most one input Message.
errorRef: Error [0..*]	This attribute specifies errors that the Operation may return. An Operation may refer to zero or more Error elements.

8.4.4. Service Package XML Schemas

Table 8-83 – interface XML schema

```
<xsd:element name="interface" type="tInterface"/>
```

```
<xsd:complexType name="tInterface">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element ref="operation" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-84 – operation XML schema

```
<xsd:element name="operation" type="tOperation"/>
<xsd:complexType name="tOperation">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
        <xsd:element name="inMessageRef" type="xsd:QName" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="outMessageRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="errorRef" type="xsd:QName" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 8-85 – endPoint XML schema

```
<xsd:element name="endPoint" type="tEndPoint"/>
<xsd:complexType name="tEndPoint">
    <xsd:complexContent>
        <xsd:extension base="tRootElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

9. Collaboration

Note: The contents of this chapter are required for BPMN **Choreography Modeling Conformance**, BPMN **Process Modeling Conformance**, or for BPMN **Complete Conformance**. However, this chapter is not required for, BPMN **Process Execution Conformance** or BPMN **BPEL Process Execution Conformance**. For more information about BPMN conformance types, see page 28.

The `Collaboration` package contains classes which are used for modeling Collaborations, which is a collection of *Participants* shown as Pools, their interactions as shown by Message Flow, and may include Processes within the Pools and/or Choreographies between the Pools (see Figure 9-1). When a Collaboration is defined it is contained in `Definitions`.

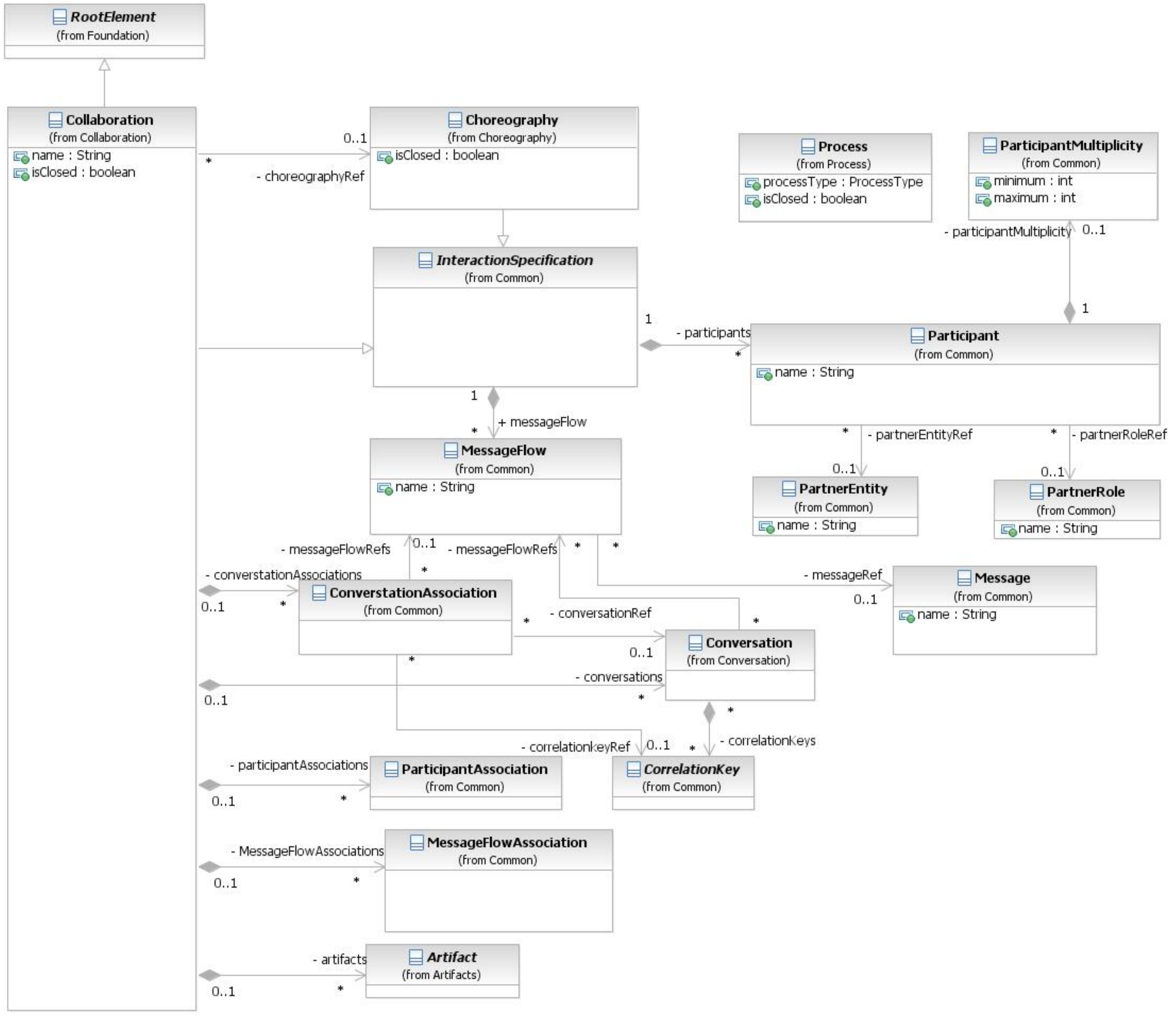


Figure 9-1 – Classes in the Collaboration package

The Collaboration element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement, and InteractionSpecification (see Table 8-48). Table 9-1 presents the additional attributes and model associations for the Collaboration element:

Table 9-1 – Collaboration Attributes and Model Associations

Attribute Name	**Description/Usage**
name: string	Name is a text description of the Collaboration.
choreographyRef: Choreography [0..1]	The choreographyRef model association defines a Choreography that is shown between the Pools of the Collaboration. A Choreography specifies a business contract (or the order in which messages will be exchanged) between interacting *Participants*. See page 327 for more details on Choreography. The participantAssociations (see below) are used to map the *Participants* of the Choreography to the *Participants* of the Collaboration. The choreographyMessageFlowAssociations (see below) are used to map the Message Flow of the Choreography to the Message Flow of the Collaboration.
conversationAssociations: ConversationAssociation [0..*]	This attribute provides the list of ConversationAssociations that are used to apply (reusable) Conversations to the Collaboration. The ConversationAssociations is used to identify the Message Flow that are grouped by the referenced Conversation. This grouping can be done automatically through the CorrelationKey of the Conversation (matching the CorrelationKey to the Messages of the Message Flow) or done through user selection if a CorrelationKey has not been defined. If the Conversation lists *Participants*, then the participantAssociations (see below) are used to map the *Participants* of the Conversation to the *Participants* of the Collaboration. If the Conversation lists Message Flow, then the MessageFlowAssociations (see below) are used to map the Message Flow of the Conversation to the Message Flow of the Collaboration.
conversations: Conversation [0..*]	The conversation model aggregation relationship allows to have Conversations contained in a Collaboration, to group Message Flow of the Collaboration and associate *correlation* information, as is required for the definitional Collaboration of a Process model. Such a Conversation SHOULD only use Message Flow references to group the Message Flow of the enclosing Collaboration.
artifacts: Artifact [0..*]	This attribute provides the list of Artifacts that are contained within the Collaboration.

participantAssociations: ParticipantAssociations [0..*]	This attribute provides a list of mappings from the *Participants* of a referenced Choreography or Conversation to the *Participants* of the Collaboration. It can also provide mappings between *Participants* of a *definitional* Collaboration for a Process to *Participants* in *definitional* Collaboration of called Processes.
messageFlowAssociations: Message Flow Association [0..*]	This attribute provides a list of mappings for the Message Flow of the Collaboration to Message Flow of a referenced model. This applies for two (2) situations: • When a Choreography is referenced by the Collaboration. • When a Conversation is referenced by the Collaboration.
IsClosed: boolean = false	A Boolean value specifying whether Message Flow not modeled in the Collaboration can occur when the Collaboration is carried out. If the value is *true*, they MAY NOT occur. If the value is *false*, they MAY occur.

A set of Messages Flow of a particular Collaboration may belong to the same Conversation. A Conversation is a set of Message Flow that share a particular purpose—i.e., they all relate to the handling of a single order (see page 112 for more information about Conversations). Correlations are the mechanism that is used to identify the Messages and, consequently, the Message Flow that belong to the same Conversation. Correlations can be used to specify Conversations between Processes that follow a fairly simple Conversation pattern in the sense that:

- The conceptual data of the Conversation is well known and defined by the participating Processes. However this doesn't mandate that underlying type systems are identical. It is sufficient that the data is known "conceptually" on a (potentially very high) business level.
- A Conversation takes place by means of simple Message exchange between Processes, no additional agreements must be considered.
- There exists send and receive Tasks accepting the conceptual data of the Conversation. (An Order send by a Task of a Process should be received by at least one Task of the participating Process)
- The *correlation* itself is defined in terms of correlation fields, which denote a subset of the conceptual data that should be used for the *correlation*. (For example, if the conceptual data comprises of an order than the correlation field might be denoted by the order ID).

In some applications it is useful to allow more Messages to be sent between *Participants* when a Collaboration is carried out than are contained the Collaboration model. This enables *Participants* to exchange other Messages as needed without changing the Collaboration. If the isClosed attribute of a Collaboration has a value of *false* or no value, then *Participants* MAY send Messages to each other without additional Message Flow in the Collaboration. If the isClosed attribute of a Collaboration has a value of *true*, then *Participants* MAY NOT send Messages to each other without additional Message Flow in the Collaboration. If a Collaboration contains a Choreography, then the value of the isClosed attribute MUST be the same in both. Restrictions on unmodeled messaging specified with isClosed apply only under the Collaboration containing the restriction. PartnerEntities and PartnerRoles of the *Participants* MAY send Messages to each other under other Choreographies, Collaborations, and Conversations.

9.1. Basic Collaboration Concepts

A Collaboration contains two (2) or more Pools, representing the *Participants* in the Collaboration. The Message exchange between the *Participants* is shown by a Message Flow that connects two (2) Pools (or the objects within the Pools). The Messages associated with the Message Flow may also be shown.

A Pool may be empty, a "black box," or main show a Process within. Choreographies may be shown "in between" the Pools as they bisect the Message Flow between the Pools. All combinations of Pools, Processes, and a Choreography are allowed in a Collaboration.

9.1.1. Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography, as well as Collaboration; they are used in these diagrams. The next few sections will describe the use of Messages, Message Flow, *Participants*, Sequence Flow, Artifacts, *Correlations*, *Expressions*, and *Services* in Choreography.

9.2. Pool and Participant

A Pool represents a Participant in a Collaboration or a Choreography. A *Participant* (see page 124) can be a specific PartnerEntity (e.g., a company) or can be a more general PartnerRole (e.g., a buyer, seller, or manufacturer). Graphically, a Pool is a container for partitioning a Process from other Pools. A Pool is not required to contain a Process, i.e., it can be a "black box".

- ◆ A Pool is a square-cornered rectangle that MUST be drawn with a solid single line (see Figure 9-2).
  - ◆ The label for the Pool MAY be placed in any location and direction within the Pool, but MUST be separated from the contents of the Pool by a single line.
    - ◆ If the Pool is a black box (i.e., does not contain a Process), then the label for the Pool MAY be placed anywhere within the Pool without a single line separator.
  - ◆ One, and only one, Pool in a diagram MAY be presented without a boundary. If there is more than one Pool in the diagram, then the remaining Pools MUST have a boundary.

The use of text, color, size, and lines for a Pool MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

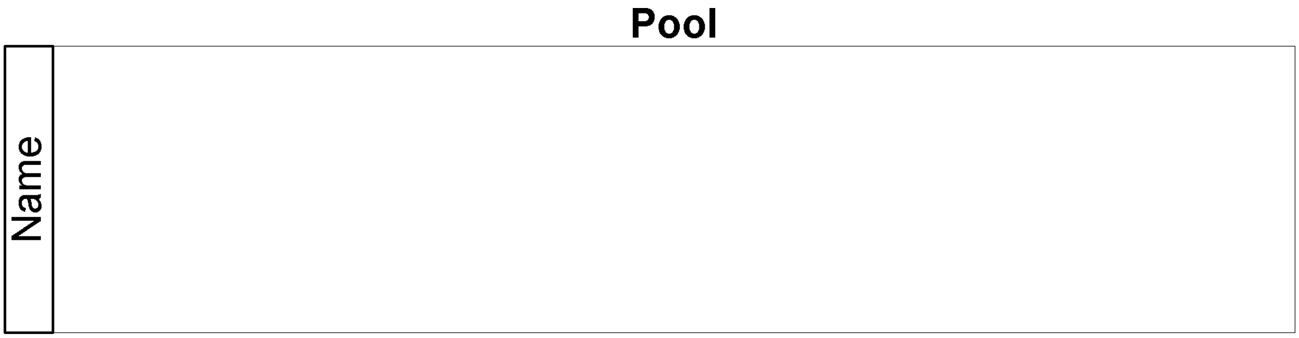


Figure 9-2 – A Pool

To help with the clarity of the Diagram, a Pool extends the entire length of the Diagram, either horizontally or vertically. However, there is no specific restriction to the size and/or positioning of a Pool. Modelers and modeling tools can use Pools in a flexible manner in the interest of conserving the "real estate" of a Diagram on a screen or a printed page.

A Pool acts as the container for the Sequence Flow between Activities (of a contained Process). The Sequence Flow can cross the boundaries between Lanes of a Pool (see page 149 for more details on Lanes), but cannot cross the boundaries of a Pool. That is, a Process is fully contained within the Pool. The interaction between Pools is shown through Message Flow.

Another aspect of Pools is whether or not there is any Activity detailed within the Pool. Thus, a given Pool may be shown as a "White Box," with all details (e.g., a Process) exposed, or as a "Black Box," with all details hidden. No Sequence Flow is associated with a "Black Box" Pool, but Message Flow can attach to its boundaries (see Figure 9-3).

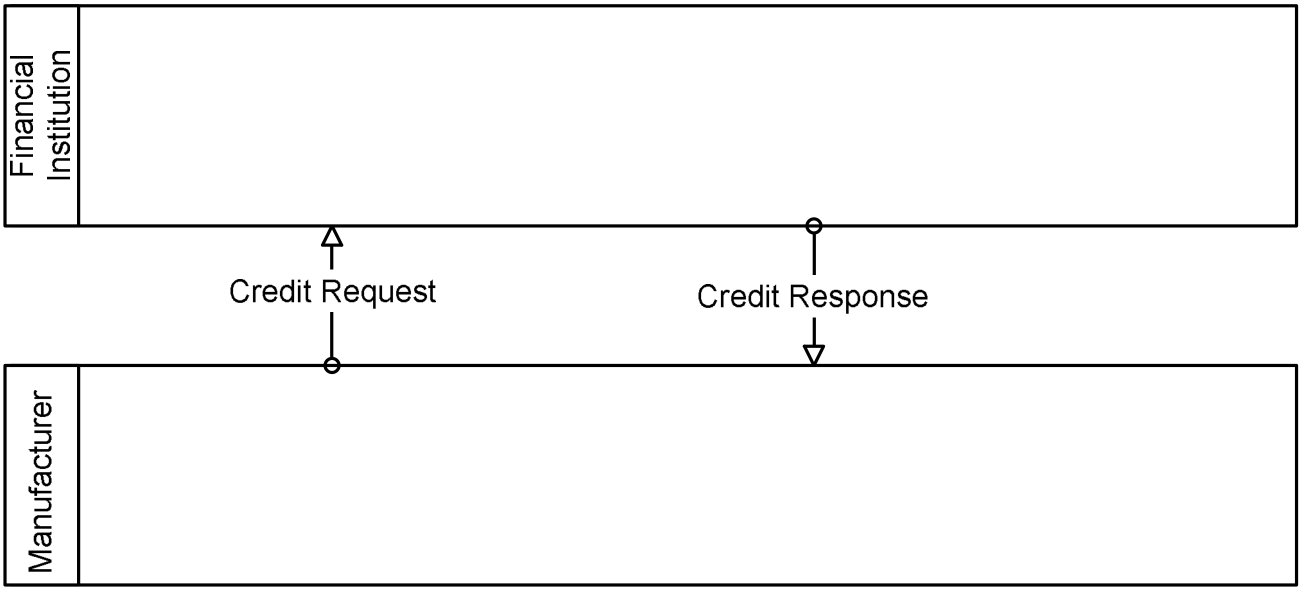


Figure 9-3 – Message Flow connecting to the boundaries of two Pools

For a "White Box" Pool, the Activities within are organized by Sequence Flow. Message Flow can cross the Pool boundary to attach to the appropriate Activity (see Figure 9-4)

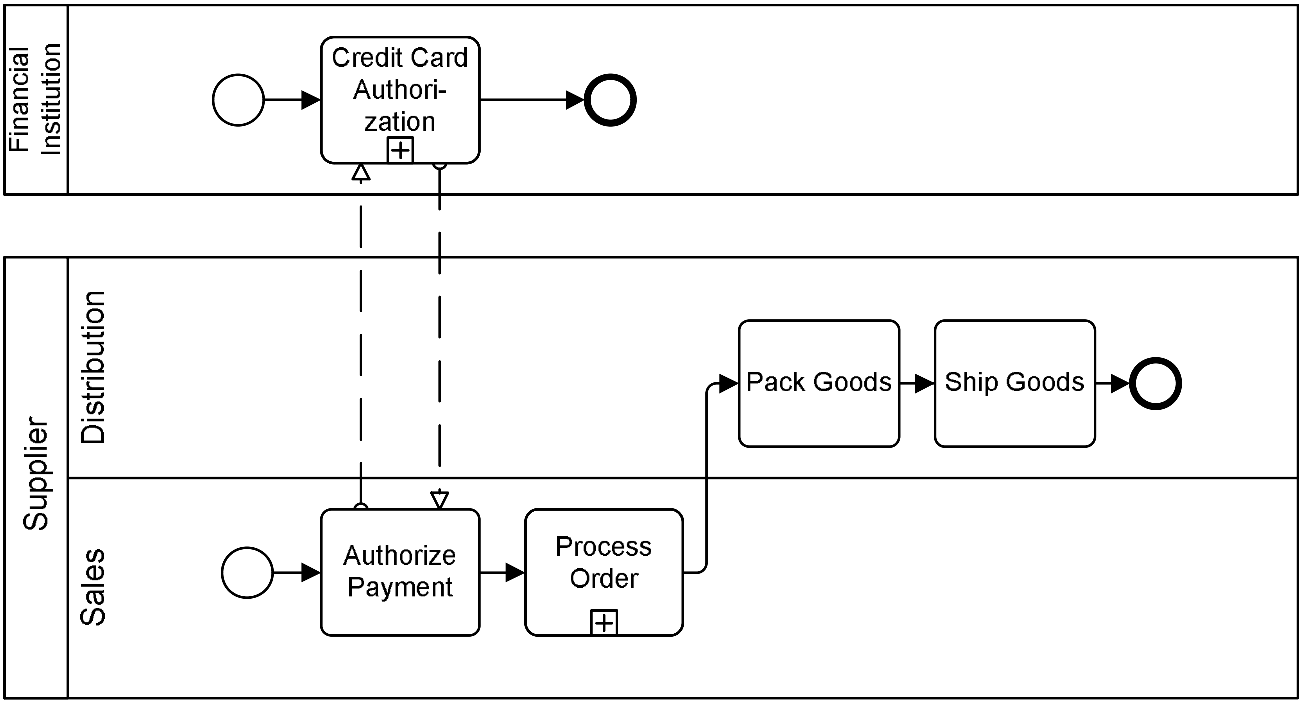


Figure 9-4 – Message Flow connecting to Flow Objects within two Pools

A Collaboration contains at least two (2) Pools (i.e., *Participants*). However, the Activities that represent the work performed from the point of view of the modeler or the modeler's organization may be considered "internal" Activities and are not required to be surrounded by the boundary of their Pool, while the other Pools in the Diagram MUST have their boundary (see Figure 9-5).

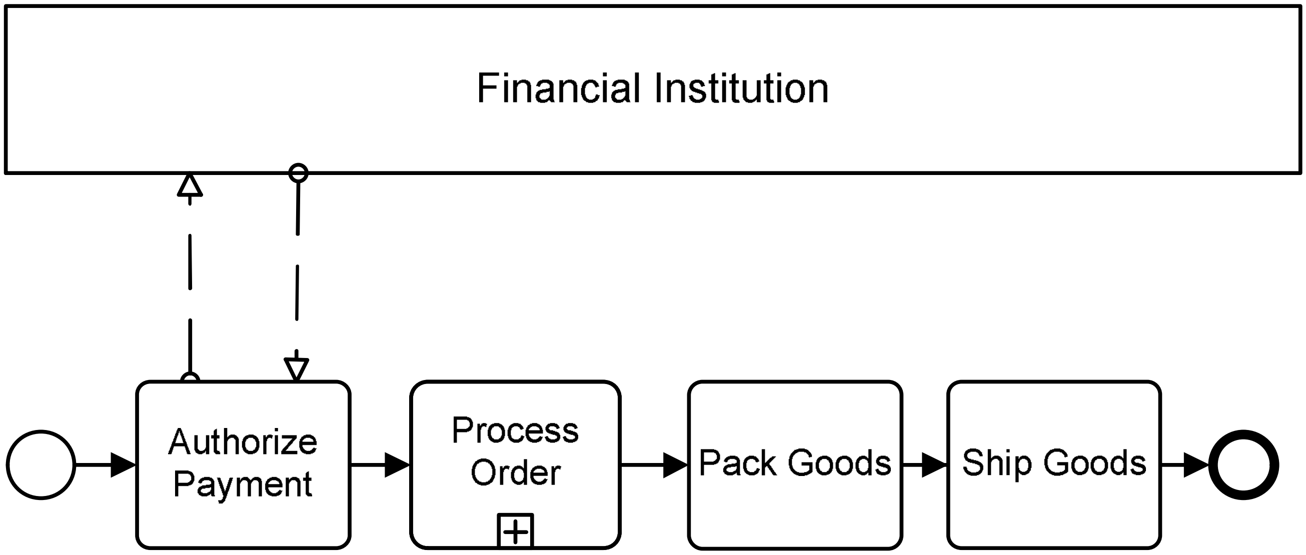


Figure 9-5 – Main (Internal) Pool without boundaries

BPMN specifies a marker for Pools: a *multi-instance* Marker May be displayed for a Pool (see Figure 9-6). The marker is used if the *Participant* defined for the Pool is a *multi-instance Participant.* See page 125 for more information on *Participant* multiplicity.

- The marker for a Pool that is a *multi-instance* MUST be a set of three vertical lines in parallel.
- The marker, if used, MUST be centered at the bottom of the shape.

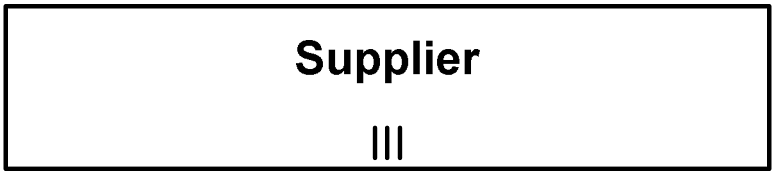


Figure 9-6 – A Pool with a Multi-Instance Participant Marker

9.2.1. Lanes

A Lane is a sub-partition within a Pool or a Process and will extend the entire length of the diagram, either vertically or horizontally. See page 316 for more information on Lanes.

9.3. Collaboration

Processes can be included in a Collaboration diagram. A *Participant*/Pool within the Collaboration can contain a Process (but they are not required). An example of this is shown in Figure 9-4, above. See page 324 for more details of how Processes are included with Collaborations.

9.4. Choreography within Collaboration

Choreographies can be included in a Collaboration diagram. A Collaboration specifies how the *Participants* and Message Flow in the Choreography are matched up with the *Participants* and Message Flow in the Collaboration. A Collaboration uses `ParticipantAssociations` and `MessageFlowAssociations` for this purpose.

To handle the *Participants*, the `innerParticipant` of a `ParticipantAssociation` refers to a *Participant* in the Choreography, while the `outerParticipant` refers to a *Participant* in the Collaboration containing the Choreography. This mapping matches the Participant Bands of the Choreography Activities in the Choreography to the Pools in the Collaboration. Thus, the names in the Participant Bands are not required (see Figure 9-7).

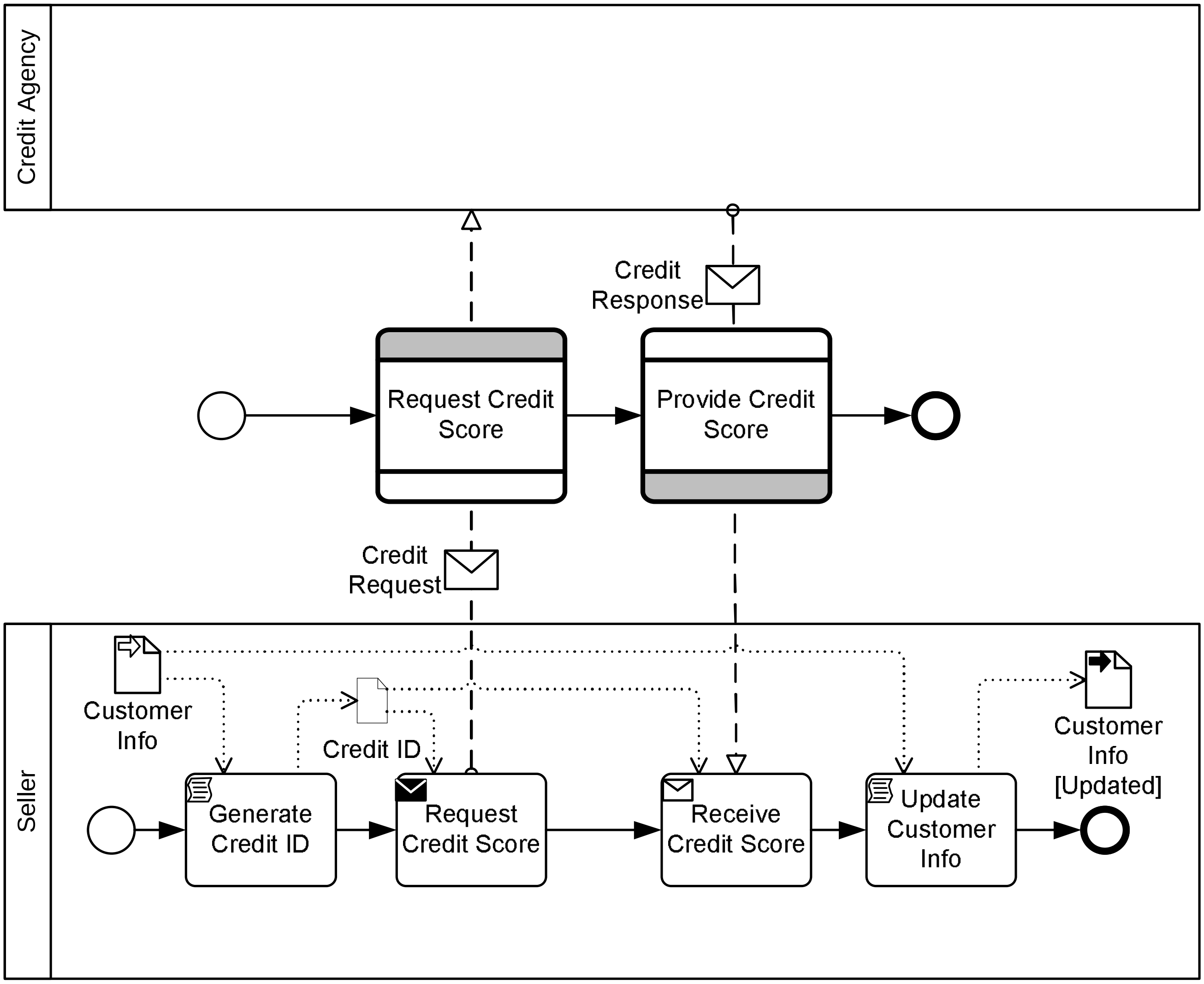


Figure 9-7 – An example of a Choreography within a Collaboration

To handle Message Flow, the innerMessageFlow of a MessageFlowAssociation refers to a Message Flow in the Choreography, while the outerMessageFlow refers to a Message Flow in the Collaboration containing the Choreography. This mapping matches the Message Flow of the Choreography (which are not visible) to the Message Flow in the Collaboration (which are visible). This allows the Message Flow of the Collaboration to be "wired up" through the appropriate Choreography Activity in the Choreography (see Figure 9-7, above).

The ParticipantAssociations might be derived from the partnerEntities or partnerRoles of the *Participants*. For example, if a Choreography Activity has a *Participant* with the same partnerEntity as a *Participant* in the Collaboration containing the Choreography, then these two (2) *Participants* could be assumed to be the inner and outerParticipants of a ParticipantAssociation. Similarly, Message Flow that reference the same Message in a Call Choreography Activity and the Collaboration, could be automatically synchronized by a MessageFlowAssociation, if only one Message Flow has that Message.

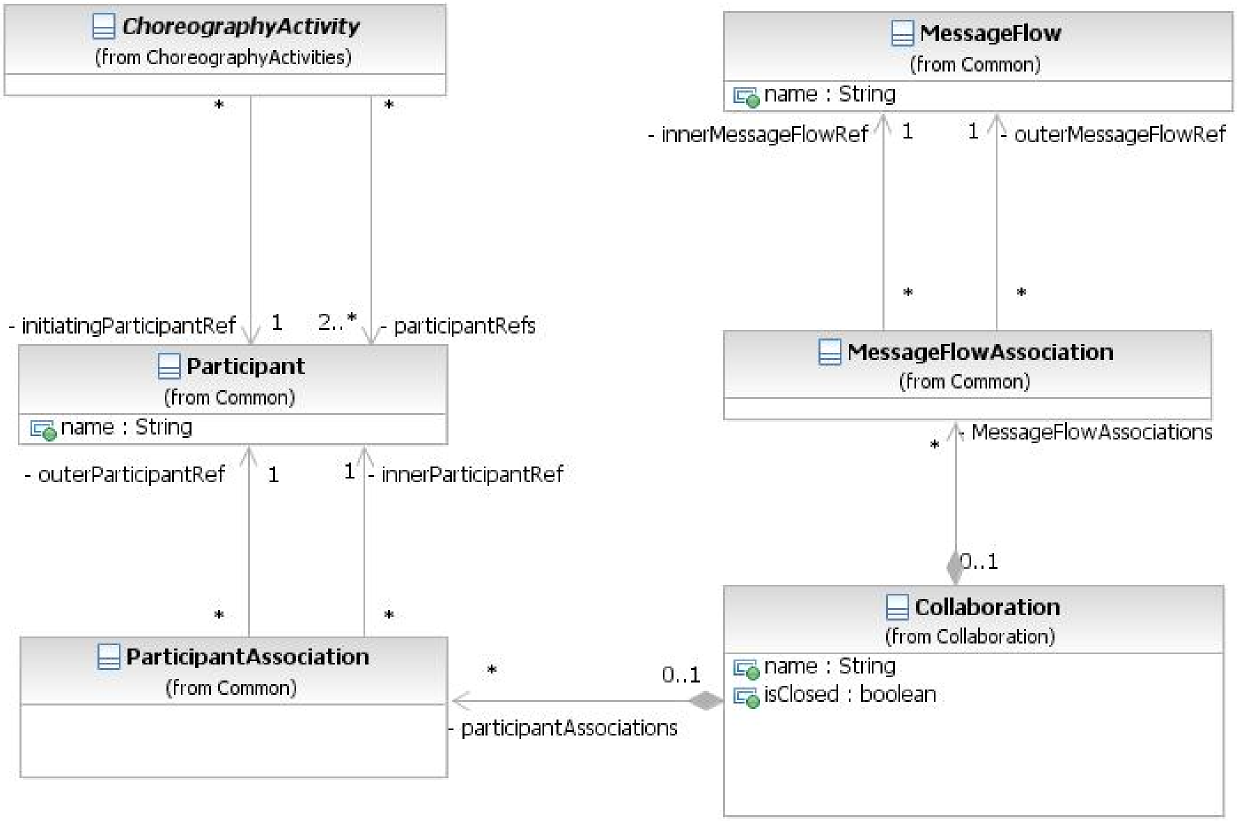


Figure 9-8 – Choreography within Collaboration class diagram

9.5. Collaboration Package XML Schemas

Table 9-2 – Collaboration XML schema

```
<xsd:element name="collaboration" type="tCollaboration" substitutionGroup="rootElement"/>
<xsd:complexType name="tCollaboration">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:sequence>
                <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="conversation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="conversationAssociation" minOccurs="0"
                        maxOccurs="unbounded"//>
                <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="MessageFlowAssociation" type="tMessageFlowAssociation"
                        minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="isClosed" type="xsd:boolean" default="false"/>
            <xsd:attribute name="choreographyRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

10. Process

Note: The content of this chapter is required for BPMN **Process Modeling Conformance** or for BPMN **Complete Conformance**. However, this chapter is not required for BPMN **Process Choreography Conformance**, BPMN **Process Execution Conformance**, or BPMN **BPEL Process Execution Conformance**. For more information about BPMN conformance types, see Page 28.

A Process describes a sequence or flow of Activities in an organization with the objective of carrying out work. In BPMN a Process is depicted as a graph of Flow Elements, which are a set of Activities, Events, Gateways, and Sequence Flow that define finite execution semantics (see Figure 10-1). Processes may be defined at any level from enterprise-wide Processes to Processes performed by a single person. Low-level Processes may be grouped together to achieve a common business goal.

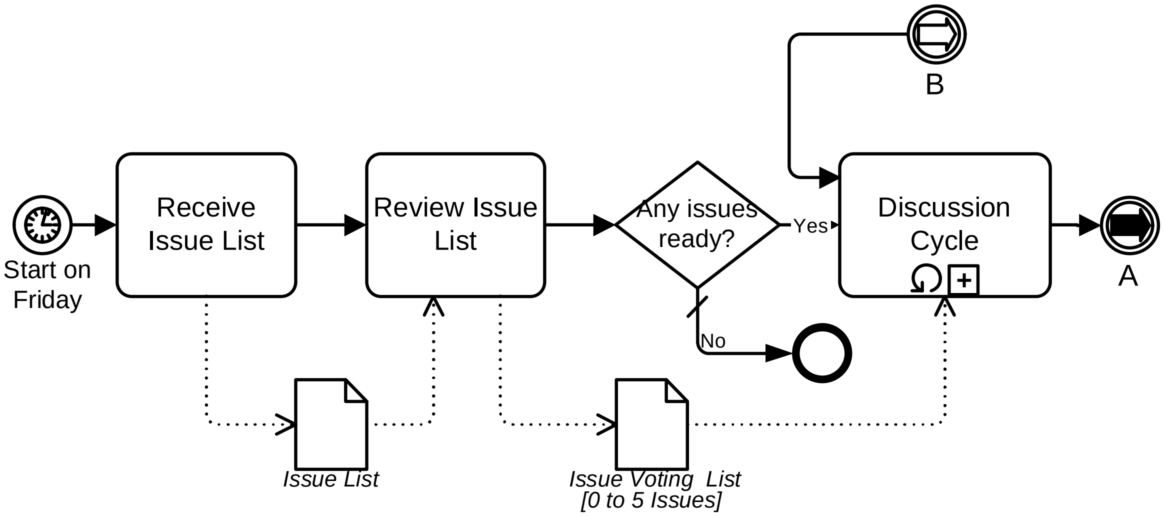


Figure 10-1 – An Example of a Process

Note that BPMN uses the term Process specifically to mean a set of *flow elements*. It uses the terms Collaboration and Choreography when modeling the interaction between Processes.

The Process package contains classes which are used for modeling the flow of Activities, Events, and Gateways, and how they are sequenced within a Process (see Figure 10-2). When a Process is defined it is contained within Definitions.

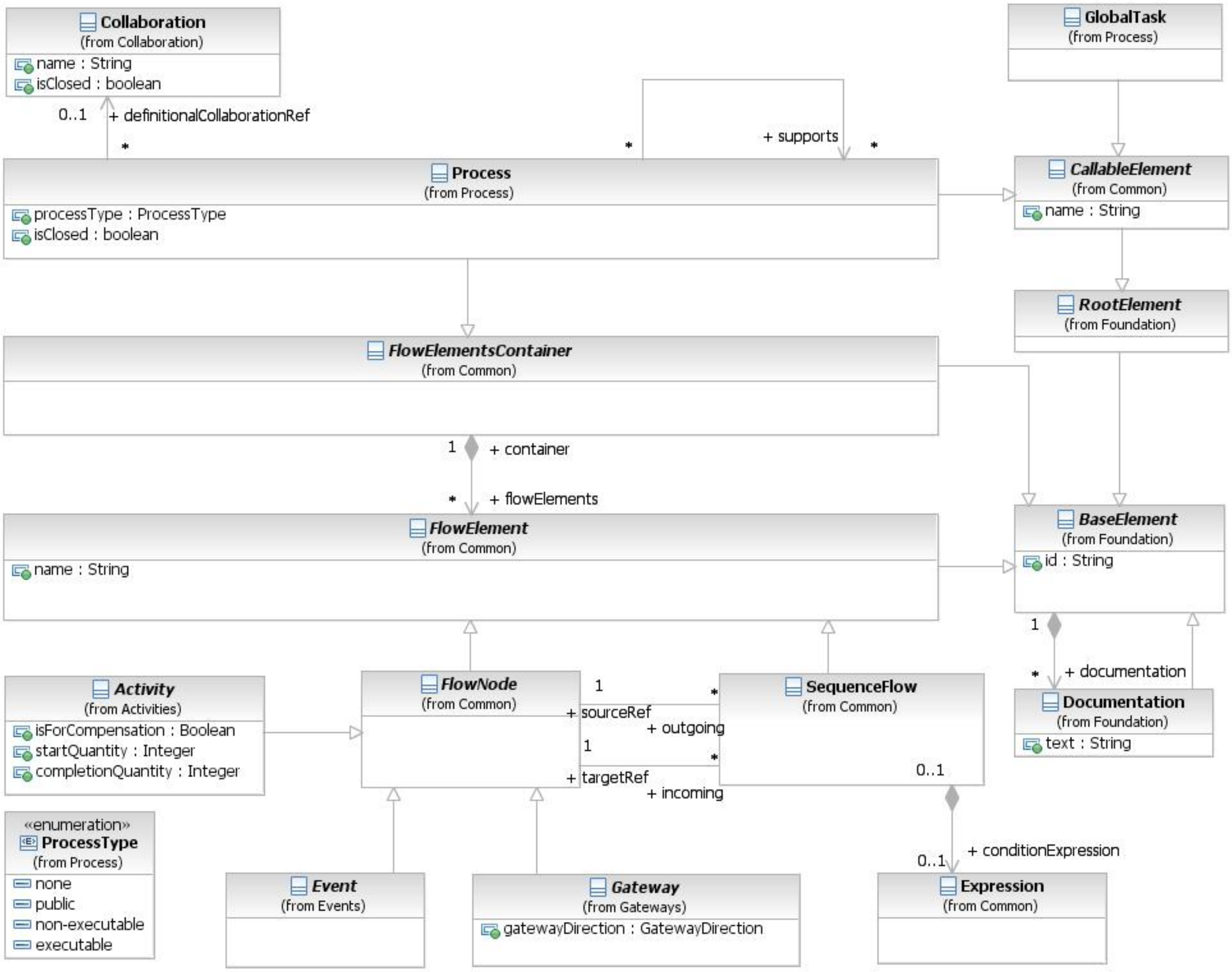


Figure 10-2 – Process class diagram

A `Process` is a `CallableElement`, allowing it to be referenced and reused by other `Processes` via the `Call Activity` construct. In this capacity, a `Process` may reference a set of `Interfaces` that define its external behavior.

A `Process` is a reusable element and can be imported and used within other `Definitions`.

Figure 10-3 shows the details of the attributes and model associations of a Process.

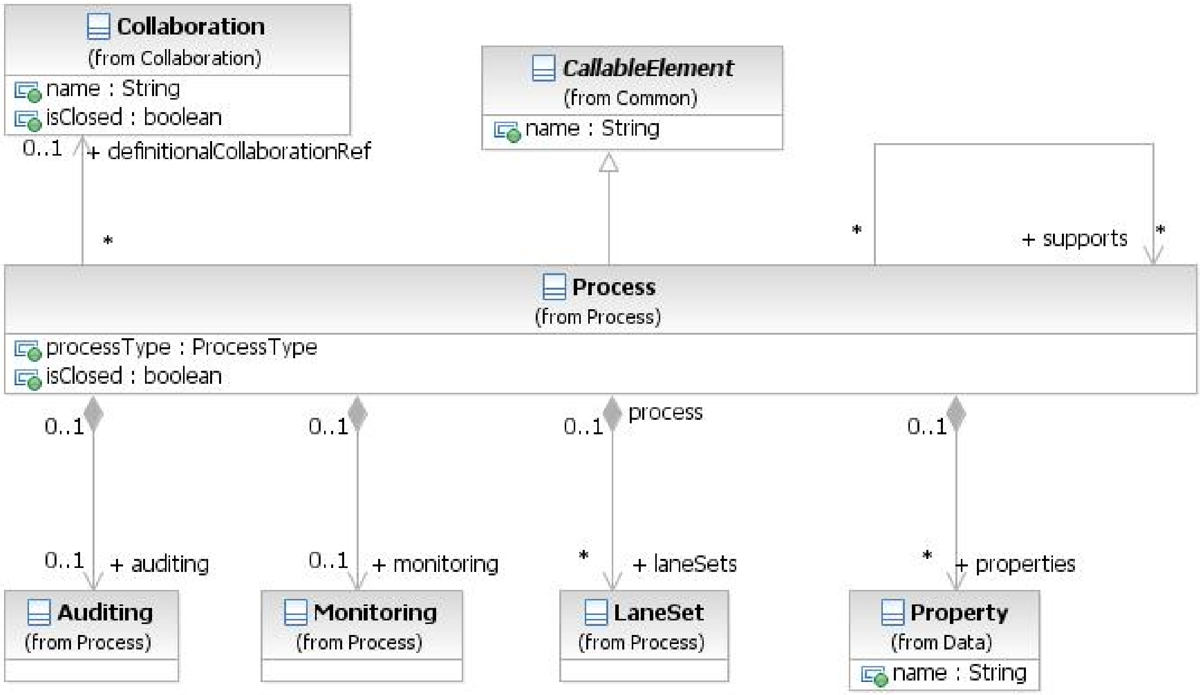


Figure 10-3 – Process Details class diagram

The Process element inherits the attributes and model associations of CallableElement (see Table 8-30) and of FlowElementContainer (see Table 8-46). Table 10-1 presents the additional attributes and model associations of the Process element:

Table 10-1 – Process Attributes & Model Associations

Attribute Name	**Description/Usage**
processType: ProcessType = none { none **|** executable **|** non-executable **|** public }	The processType attribute Provides additional information about the level of abstraction modeled by this Process. A *public* Process shows only those flow elements that are relevant to external consumers. Internal details are not modeled. These Processes are publicly visible and can be used within a Collaboration. Note that the *public* processType was named *abstract* in BPMN 1.2. The BPMN 1.2 processType *private* has been divided into two (2) types: An *executable* Process is a *private* Process that has been modeled for the purpose of being executed according to the semantics of Chapter 14 (see page 426). Of course, during the development cycle of the Process, there will be stages where the Process does not have enough detail to be “executable.” A non-executable Process is a *private* Process that has been modeled for the purpose of documenting Process behavior at a modeler-defined level of detail. Thus, information required for execution, such as formal condition expressions are typically not included in a *non-executable* Process. By default, the processType is “none”, meaning undefined.
auditing: Auditing [0..1]	This attribute provides a hook for specifying audit related properties.
monitoring: Monitoring [0..1]	This attribute provides a hook for specifying monitoring related properties.
laneSets: LaneSet [0..*]	This attribute defines the list of LaneSets used in the Process.
IsClosed: boolean = false	A Boolean value specifying whether interactions, such as sending and receiving Messages and Events, not modeled in the Process can occur when the Process is executed or performed. If the value is *true*, they MAY NOT occur. If the value is *false*, they MAY occur.
supports: Process [0..*]	Modelers can declare that they intend all executions or performances of one Process to also be valid for another Process. This means they expect all the executions or performances of the first Processes to also follow the steps laid out in the second Process.

properties: Property [0..*]	Modeler-defined properties MAY be added to a Process. These properties are "local" to the Process. All Tasks and Sub-Processes SHALL have access to these properties. The fully delineated name of these properties is "<process name>.<property name>" (e.g., "Add Customer.Customer Name"). If a Process is embedded within another Process, then the fully delineated name SHALL also be preceded by the *parent* Process name for as many *parents* there are until the top level Process.
definitionalCollaborationRef: Collaboration [0..1]	For Processes that interact with other *Participants*, a definitional Collaboration can be referenced by the Process. The definitional Collaboration specifies the *Participants* the Process interacts with, and more specifically, which individual service, Send or Receive Task, or Message Event, is connected to which *Participant* through Message Flow. The definitional Collaboration need not be displayed. Additionally, the definitional Collaboration can be used to include Conversation information within a Process.

In addition, a Process *Instance* has attributes whose values may be referenced by expressions (see Table 10-2). These values are only available when the Process is being executed.

Table 10-2 – Process Instance Attributes

Attribute Name	Description/Usage
state: String = inactive {inactive | ready | withdrawn | active | terminated | failed| completing | completed | compensating | compensated | closed}	The current state of this Process instance.

10.1. Basic Process Concepts

10.1.1. Types of BPMN Processes

Business Process modeling is used to communicate a wide variety of information to a wide variety of audiences. BPMN is designed to cover many types of modeling and allows the creation of end-to-end Business Processes. There are three basic types of BPMN Processes:

- *Private Non-executable* (*internal*) Business Processes
- *Private Executable* (internal) Business Processes
- *Public* Processes

Private (Internal) Business Processes

Private Business Processes are those internal to a specific organization. These Processes have been generally called workflow or BPM Processes (see Figure 10-4). Another synonym typically used in the Web services area is the *Orchestration* of services. There are two (2) types of *private* Processes: *executable* and *non-executable*. An *executable* Process is a Process that has been modeled for the purpose of being executed according to the semantics defined in Chapter 14 (see page 426). Of course, during the development cycle of the Process, there will be stages where the Process does not have enough detail to be "executable." A non-executable Process is a *private* Process that has been modeled for the purpose of documenting Process behavior at a modeler-defined level of detail. Thus, information required for execution, such as formal condition expressions are typically not included in a *non-executable* Process.

If a swimlanes-like notation is used (e.g., a Collaboration, see below) then a *private* Business Process will be contained within a single Pool. The Process flow is therefore contained within the Pool and cannot cross the boundaries of the Pool. The flow of Messages can cross the Pool boundary to show the interactions that exist between separate *private* Business Processes.

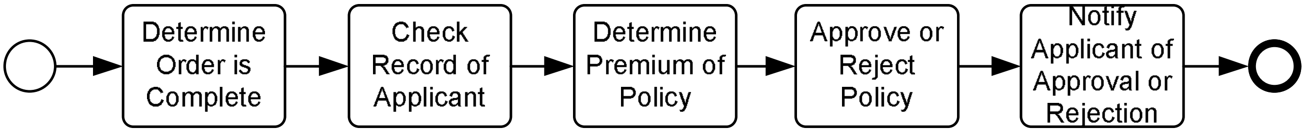

Determine Order is Complete
Check Record of Applicant
Determine Premium of Policy
Approve or Reject Policy
Notify Applicant of Approval or Rejection

Figure 10-4 – Example of a private Business Process

Public Processes

A *public* Process represents the interactions between a *private* Business Process and another Process or *Participant* (see Figure 10-5). Only those Activities that are used to communicate to the other *Participant(s)*, plus the order of these Activities, are included in the *public* Process. All other "internal" Activities of the *private* Business Process are not shown in the *public* Process. Thus, the *public* Process shows to the outside world the Messages, and the order of these Messages, that are required to interact with that Business Process. *Public* Processes can be modeled separately or within a Collaboration to show the flow of Messages between the *public* Process Activities and other *Participants*. Note that the *public* type of Process was named "abstract" in BPMN 1.2.

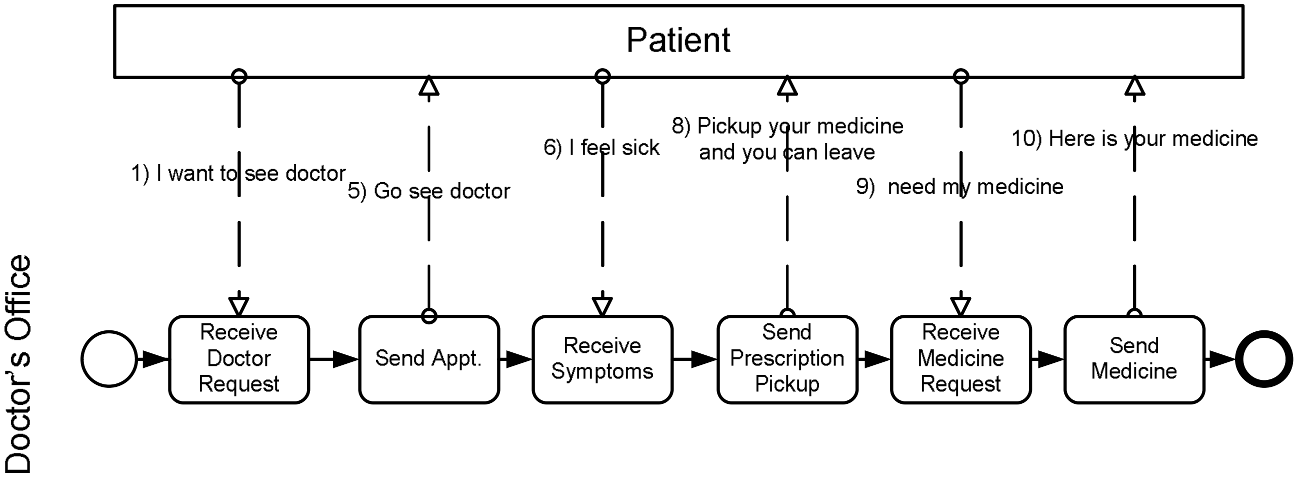


Figure 10-5 – Example of a *public* Process

10.1.2. Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography, as well as Collaboration; they are used in both types of diagrams. The next few sections will describe the use of Messages, Message Flow, *Participants*, Sequence Flow, Artifacts, *Correlations*, *Expressions*, and *Services* in Choreography.

The key graphical elements of Gateways and Events are also common to both Choreography and Process. Since their usage has a large impact, they are described in major sections of this chapter (see page 239 for Events and page 295 for Gateways).

10.2. Activities

An Activity is work that is performed within a Business Process. An Activity can be atomic or non-atomic (compound). The types of Activities that are a part of a Process are: Task, Sub-Process, and Call Activity, which allows the inclusion of re-usable Tasks and Processes in the diagram. However, a Process is not a specific graphical object. Instead, it is a set of graphical objects. The following sections will focus on the graphical objects Sub-Process and Task.

Activities represent points in a Process flow where work is performed. They are the executable elements of a BPMN Process.

The Activity class is an abstract element, sub-classing from FlowElement (as shown in Figure 10-6).

Concrete sub-classes of Activity specify additional semantics above and beyond that defined for the generic Activity.

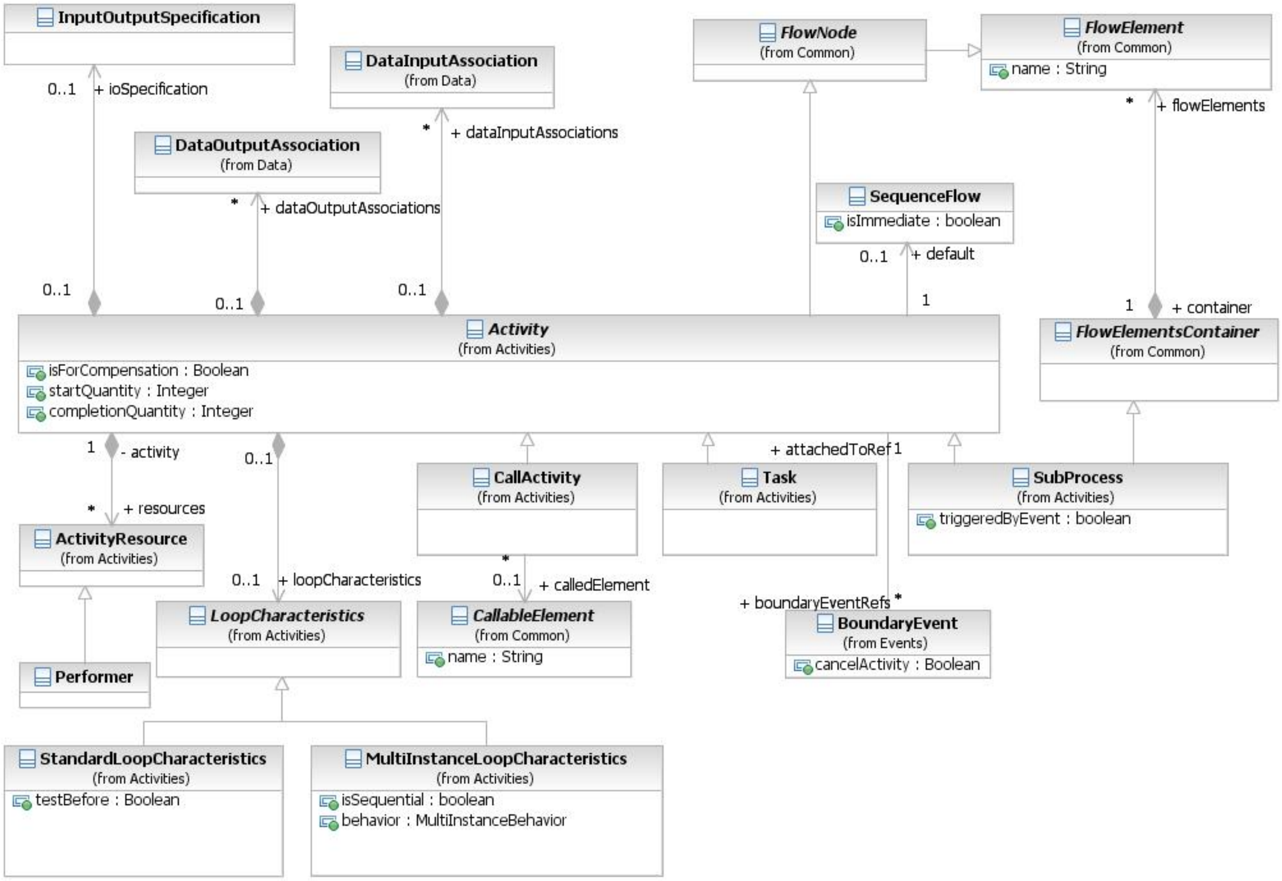


Figure 10-6 – Activity class diagram

The Activity class is the abstract super class for all concrete Activity types.

The Activity element inherits the attributes and model associations of FlowElement (see Table 8-45). Table 10-3 presents the additional attributes and model associations of the Activity element:

Table 10-3 – Activity attributes and model associations

Attribute Name	Description/Usage
isForCompensation: boolean = false	A flag that identifies whether this activity is intended for the purposes of *compensation*. If false, then this activity executes as a result of normal execution flow. If true, this activity is only activated when a Compensation Event is detected and initiated under Compensation Event visibility *scope* (see page 288 for more information on *scopes*).
loopCharacteristics: LoopCharacteristics [0..1]	An activity may be performed once or may be repeated. If repeated, the activity MUST have loopCharacteristics which define the repetition criteria (if the processType attribute of the Process is set to executable).

resources: ActivityResource [0..*]	Defines the resource that will perform or will be responsible for the activity. The resource, e.g. a performer, can be specified in the form of a specific individual, a group, an organization role or position, or an organization.
default: SequenceFlow [0..1]	The Sequence Flow that will receive a *Token* when none of the conditionExpressions on other *outgoing* Sequence Flow evaluate to *true*. The *default* Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.
ioSpecification: InputOutputSpecification [0..1]	The InputOutputSpecification defines the *inputs* and *outputs* and the InputSets and OutputSets for the Activity. See page 218 for more information on the InputOutputSpecification.
properties: Property [0..*]	Modeler-defined properties MAY be added to an Activity. These properties are “local” to the Activity. The fully delineated name of these properties is “<activity name>.<property name>” (e.g., “Add Customer.Customer Name”).
boundaryEventRefs: BoundaryEvent [0..*]	This references the Intermediate Events that are attached to the boundary of the Activity.
dataInputAssociations: DataInputAssociation [0..*]	An optional reference to the DataInputAssociations. A DataInputAssociation defines how the DataInput of the Activity’s InputOutputSpecification will be populated.
dataOutputAssociations: DataOutputAssociation [0..*]	An optional reference to the DataOutputAssociations..
startQuantity: integer = 1	The default value is 1. The value MUST NOT be less than 1. This attribute defines the number of *tokens* that must arrive before the Activity can begin. Note that any value for the attribute that is greater than 1 is an advanced type of modeling and should be used with caution.
completionQuantity: integer = 1	The default value is 1. The value MUST NOT be less than 1. This attribute defines the number of *tokens* that must be generated from the Activity. This number of *tokens* will be sent done any *outgoing* Sequence Flow (assuming any Sequence Flow Conditions are satisfied). Note that any value for the attribute that is greater than 1 is an advanced type of modeling and should be used with caution.

In addition, an Activity *instance* has attributes whose values may be referenced by expressions. These values are only available when the Activity is being executed.

Table 8-57 presents the Instance attributes of the Activity element:

Table 10-4 – Activity *instance* attributes

Attribute Name	Description/Usage
state: string = none {none | ready | active | cancelled | aborting | aborted | completing | completed}	The current state of this activity instance.

Sequence Flow Connections

See Section “Sequence Flow Connections Rules” on page 64 for the entire set of objects and how they may be source or targets of Sequence Flow.

- An Activity MAY be a target for Sequence Flow; it can have multiple *incoming* Sequence Flow. *Incoming* Sequence Flow MAY be from an alternative path and/or parallel paths.
    - If the Activity does not have an *incoming* Sequence Flow, and there is no Start Event for the Process, then the Activity MUST be instantiated when the Process is instantiated.
        - There are two (2) exceptions to this: Compensation Activities and Event Sub-Processes.

Note – If the Activity has multiple *incoming* Sequence Flow, then this is considered uncontrolled flow. This means that when a *token* arrives from one of the Paths, the Activity will be instantiated. It will not wait for the arrival of *tokens* from the other paths. If another *token* arrives from the same path or another path, then a separate *instance* of the Activity will be created. If the flow needs to be controlled, then the flow should converge on a Gateway that precedes the Activities (see 295 for more information on Gateways).

- An Activity MAY be a source for Sequence Flow; it can have multiple *outgoing* Sequence Flow. If there are multiple *outgoing* Sequence Flow, then this means that a separate parallel path is being created for each Sequence Flow (i.e., *tokens* will be generated for each *outgoing* Sequence Flow from the Activity).
    - If the Activity does not have an *outgoing* Sequence Flow, and there is no End Event for the Process, then the Activity marks the end of one or more paths in the Process. When the Activity ends and there are no other parallel paths active, then the Process MUST be completed.
        - There are two (2) exceptions to this: Compensation Activities and Event Sub-Processes.

Message Flow Connections

See Section "Message Flow Connection Rules" on page 65 for the entire set of objects and how they may be source or targets of Message Flow.

Note – All Message Flow must connect two separate Pools. They can connect to the Pool boundary or to Flow Objects within the Pool boundary. They cannot connect two objects within the same Pool.

- An Activity MAY be the target for Message Flow; it can have zero or more *incoming* Message Flow.
- A Activity MAY be a source for Message Flow; it can have zero or more *outgoing* Message Flow.

10.2.1. Resource Assignment

The following sections define how required Resources can be defined for an Activity. Figure 10-7 displays the class diagram for the BPMN elements used for Resource assignment.

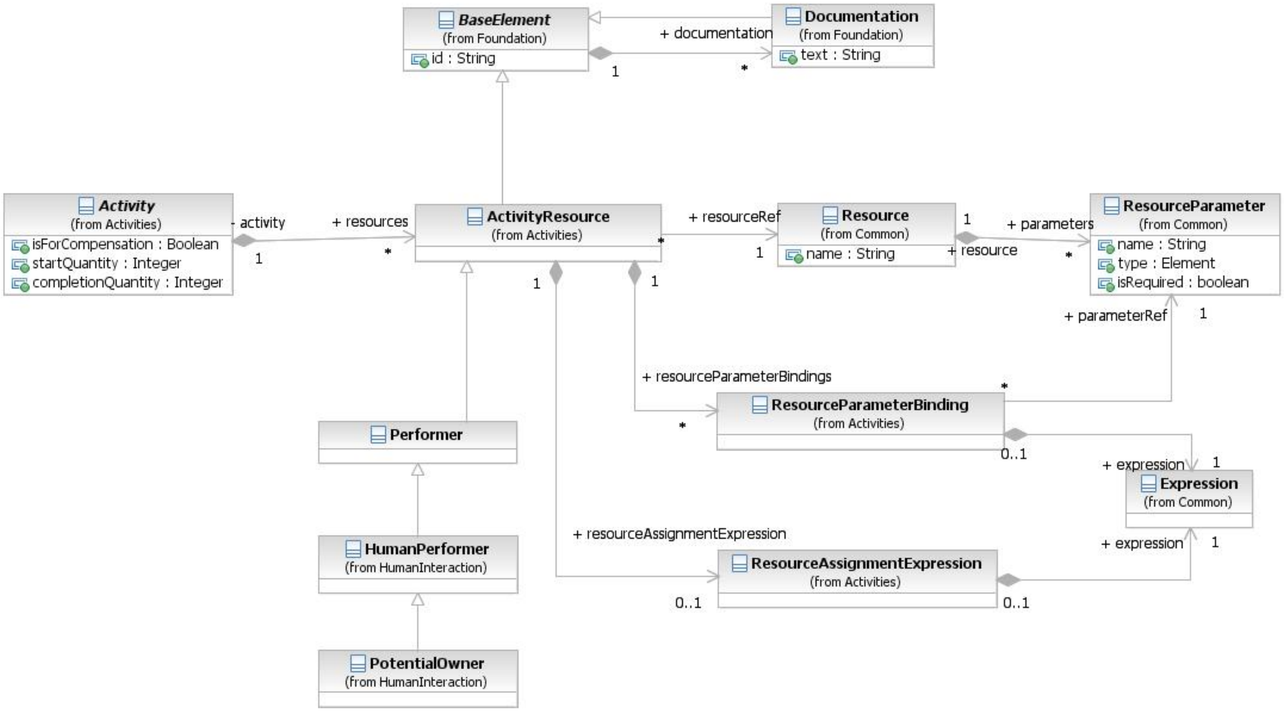


Figure 10-7 – The class diagram for assigning Resources

Activity Resource

The ActivityResource element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-5 presents the additional model associations of the ActivityResource element:

Table 10-5 – ActivityResource model associations

Attribute Name	Description/Usage
resourceRef: Resource	The Resource that will be used by the ActivityResource.
resourceAssignmentExpression: ResourceAssignmentExpression [0..1]	This defines the Expression used for the Resource assignment (see below).
resourceParameterBindings: ResourceParameterBindings [0..*]	This defines the Parameter bindings used for the Resource assignment (see below).

Expression Assignment

Resources can be assigned to an Activity using Expressions. These Expressions must return Resource entity related data types, like Users or Groups. Different Expressions can return multiple Resources. All of them are assigned to the respective subclass of the ActivityResource element, for example as potential owners. The semantics is defined by the subclass.

The ResourceAssignmentExpression element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-6 presents the additional model associations of the ResourceAssignmentExpression element:

Table 10-6 – ResourceAssignmentExpression model associations

Attribute Name	Description/Usage
expression: Expression	The element ResourceAssignmentExpression must contain an expression which is used at runtime to assign resource(s) to a ActivityResource element.

Parameterized Resource Assignment

Resources support query parameters which are passed to the Resource query at runtime. Parameters may refer to Task *instance* data using expressions. During Resource query execution, an infrastructure may decide which of the Parameters defined by the Resource are used. It may use zero (0) or more of the Parameters specified. It may also override certain Parameters with values defined during Resource deployment. The deployment mechanism for Tasks and Resources is out of scope for this specification. Resource queries are evaluated to determine the set of Resources, e.g. people, assigned to the Activity. Failed Resource queries are treated like Resource queries that return an empty result set. Resource queries return one Resource or a set of Resources.

The ResourceParameterBinding element inherits the attributes and model associations of BaseElement (see Table 8-5) Table 10-7 presents the additional model associations of the ResourceParameterBinding element:

Table 10-7 – ResourceParameterBinding model associations

Attribute Name	Description/Usage
parameterRef: ResourceParameter [1]	Reference to the parameter defined by the Resource
expression: Expression	The Expression that evaluates the value used to bind the ResourceParameter

10.2.2. Performer

The Performer class defines the resource that will perform or will be responsible for an Activity. The performer can be specified in the form of a specific individual, a group, an organization role or position, or an organization.

The Performer element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to ActivityResource, but does not have any additional attributes or model associations.

10.2.3. Tasks

A Task is an *atomic* Activity within a Process flow. A Task is used when the work in the Process cannot be broken down to a finer level of detail. Generally, an end-user and/or applications are used to perform the Task when it is executed.

A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners (see Figure 10-8).

- A Task is a rounded corner rectangle that MUST be drawn with a single thin line.
  - The use of text, color, size, and lines for a Task MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

- A boundary drawn with a thick line SHALL be reserved for Call Activity (Global Tasks) (see page 196).
- A boundary drawn with a dotted line SHALL be reserved for Event Sub-Processes (see page 188) and thus are not allowed for Tasks.
- A boundary drawn with a double line SHALL be reserved for Transaction Sub-Processes (see page 190) and thus are not allowed for Tasks.

Figure 10-8 – A Task object

BPMN specifies three types of markers for Task: a Loop marker or a Multi-Instance marker and a Compensation marker. A Task may have one or two of these markers (see Figure 10-9).

- The marker for a Task that is a standard *loop* MUST be a small line with an arrowhead that curls back upon itself. See page 200 for more information on *loop* Activities.
  - The *loop* Marker MAY be used in combination with the *compensation* marker.
- The marker for a Task that is a *multi-instance* MUST be a set of three vertical lines. See page 201 for more information on *multi-instance* Activities.
  - If the *multi-instance instances* are set to be performed in sequence rather than parallel, then the marker will be rotated 90 degrees (see Figure 10-45, below)
  - The *multi-instance* marker MAY be used in combination with the *compensation* marker.
- The marker for a Task that is used for *compensation* MUST be a pair of left facing triangles (like a tape player "rewind" button). See page 314 for more information on *compensation*.
  - The Compensation Marker MAY be used in combination with the *loop* marker or the *multi-instance* marker.
- All the markers that are present MUST be grouped and the whole group centered at the bottom of the shape.

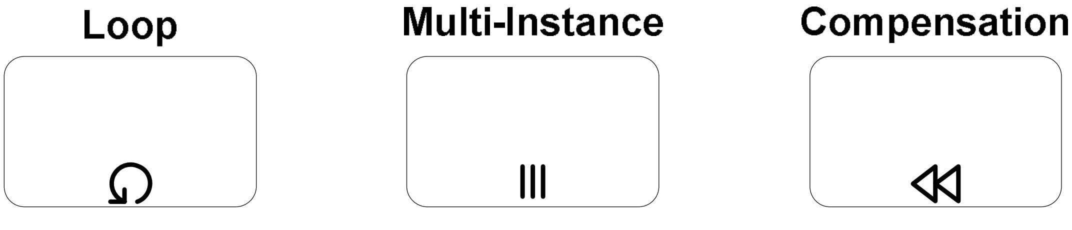


Figure 10-9 – Task markers

Figure 10-10 displays the class diagram for the Task element.

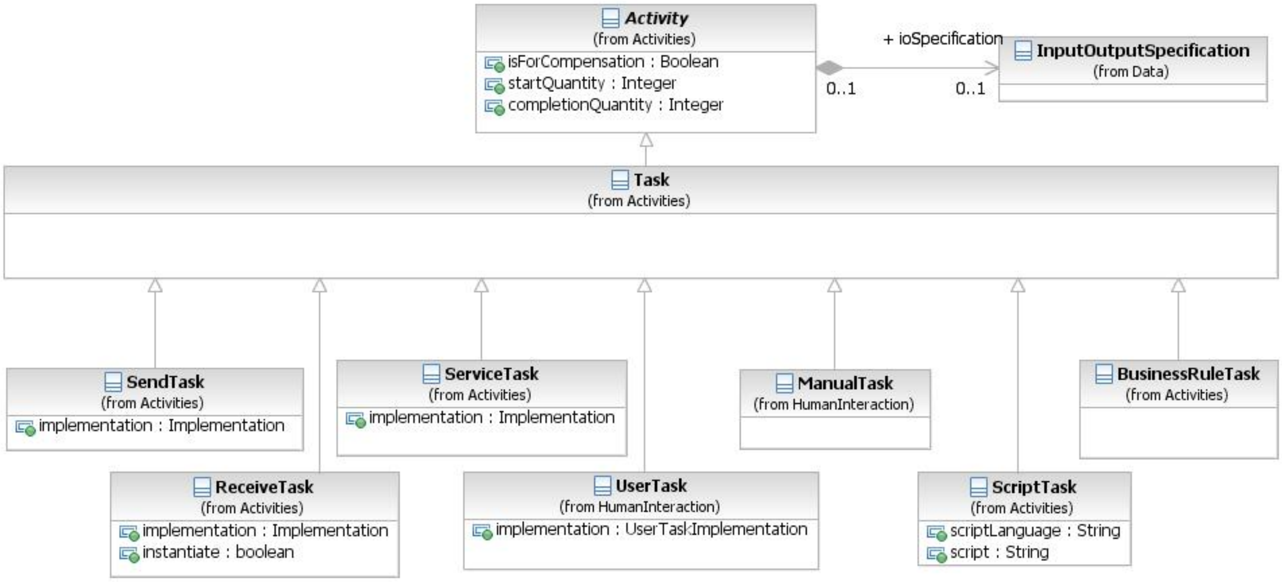


Figure 10-10 – The Task class diagram

The Task inherits the attributes and model associations of Activity (see Table 10-3). There are no further attributes or model associations of the Task.

Types of Tasks

There are different types of Tasks identified within BPMN to separate the types of inherent behavior that Tasks might represent. The list of Task types may be extended along with any corresponding indicators. A Task which is not further specified is called Abstract Task (this was referred to as the None Task in BPMN 1.2). The notation of the Abstract Task is shown in Figure 10-8.

Service Task

A Service Task is a Task that uses some sort of service, which could be a Web service or an automated application.

A Service Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the Task is a Service Task (see Figure 10-11).

- A Service Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other Task types (as shown in Figure 10-11).

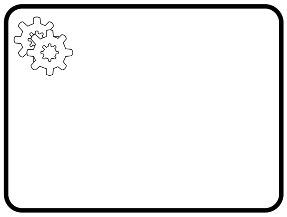


Figure 10-11 – A Service Task Object

The Service Task inherits the attributes and model associations of Activity (see Table 10-3). In addition the following constraints are introduced. The Service Task has exactly one InputSet and at most one OutputSet. The Service Task inputs map to the parts of the input Message, that is the attributes inside of the Message (see section on Operation below). For a WSDL message, this would be expressed as message parts. The Service Task outputs map to the parts of the output Message, which are the attributes inside of the Message.

The actual *Participant* whose service is used can be identified by connecting the Service Task to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.

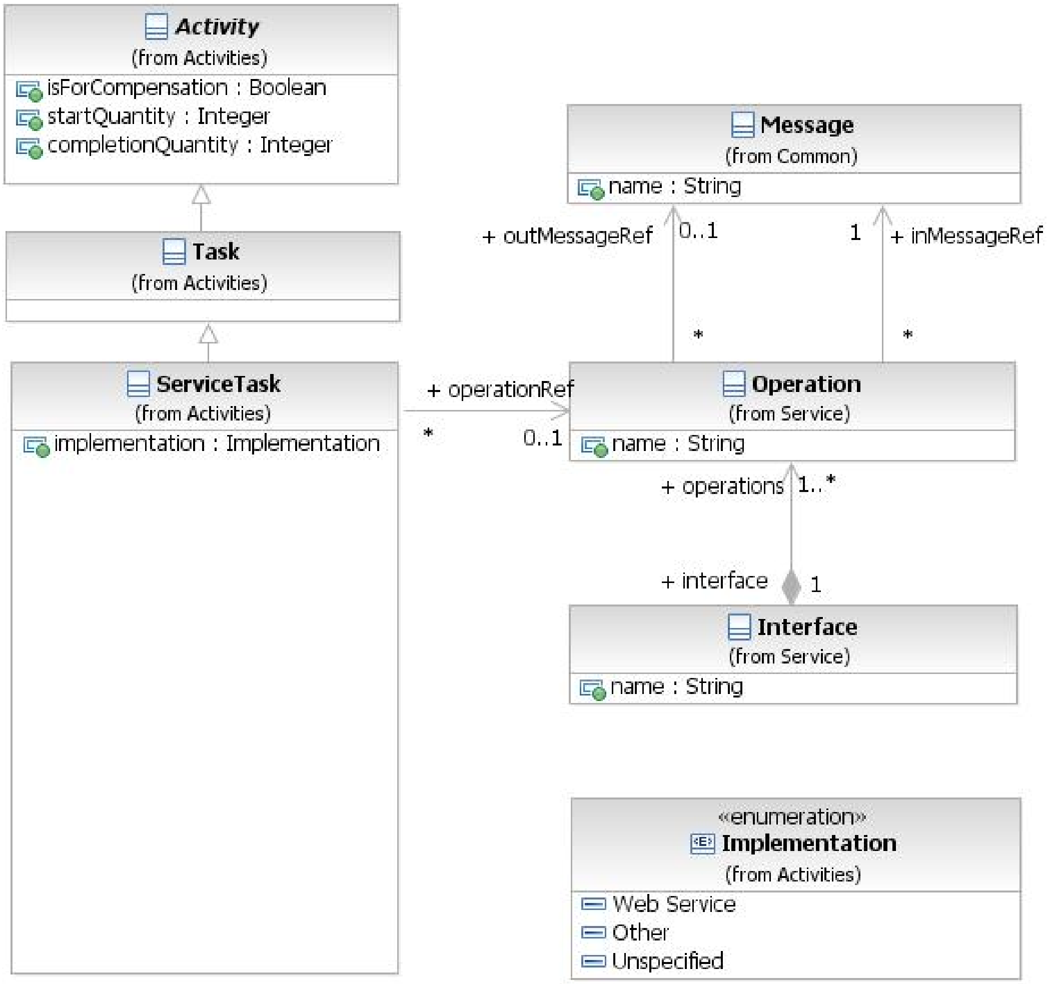


Figure 10-12 – The Service Task class diagram

The Service Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-8 presents additional the model associations of the Service Task:

Table 10-8 – Service Task model associations

Attribute Name	Description/Usage
implementation: Implementation = Web Service {Web Service | Other | Unspecified}	This attribute specifies the technology that will be used to send and receive the Messages. A Web service is the default technology.
operationRef: Operation [0..1]	This attribute specifies the operation that is invoked by the Service Task.

Send Task

A Send Task is a simple Task that is designed to send a Message to an external Participant (relative to the Process). Once the Message has been sent, the Task is completed.

The actual *Participant* which the Message is sent can be identified by connecting the Send Task to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.

A Send Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is a filled envelope marker (the same marker as a *throw* Message Event) in the upper left corner of the shape that indicates that the Task is a Send Task (see Figure 10-11).

- ◆ A Send Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a filled envelope marker that distinguishes the shape from other Task types (as shown in Figure 10-13).

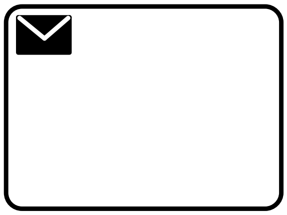


Figure 10-13 – A Send Task Object

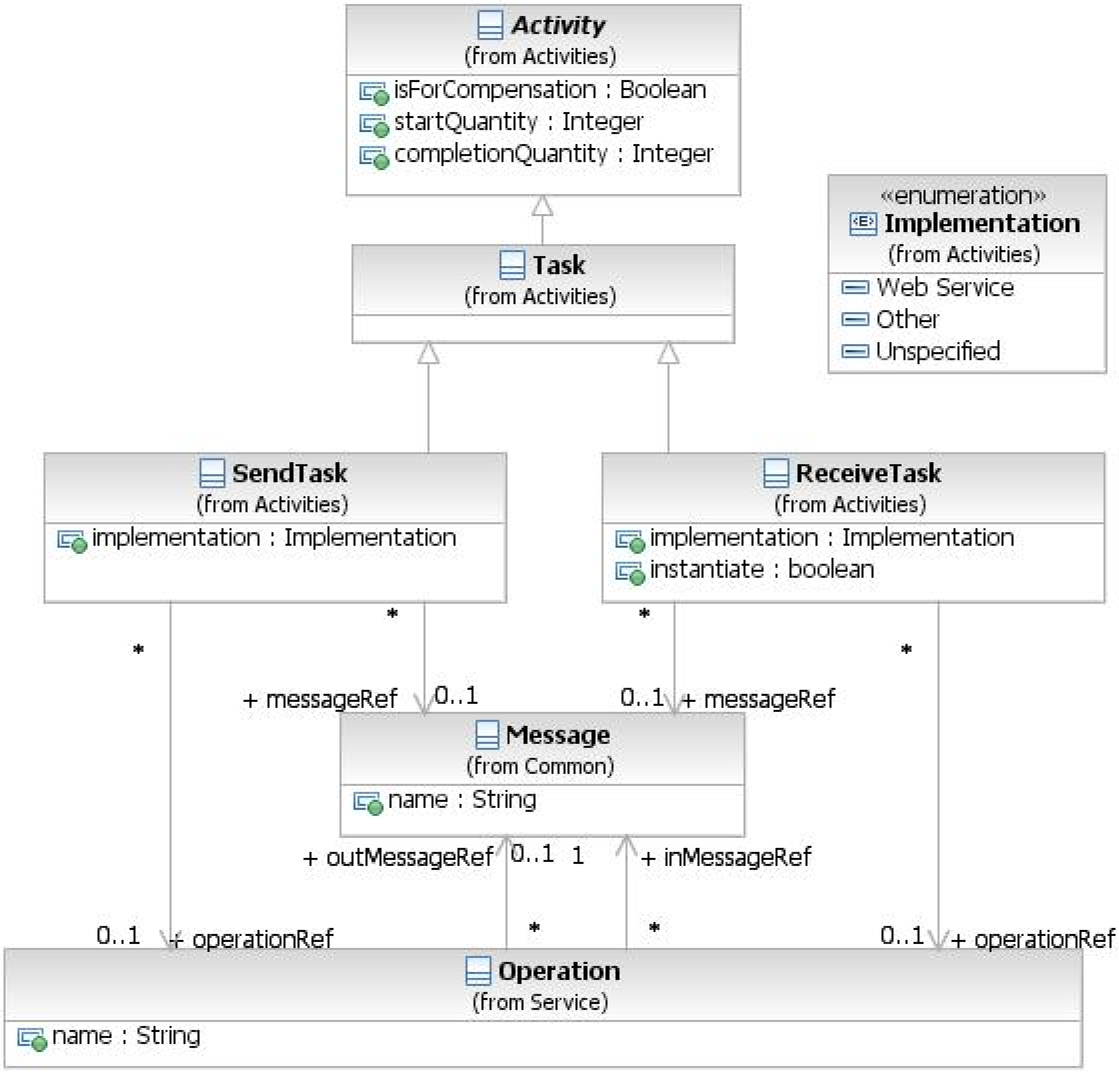


Figure 10-8 – The Send Task and Receive Task class diagram

The Send Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-9 presents the additional model associations of the Receive Task:

Table 10-9 – Send Task model associations

Attribute Name	Description/Usage
messageRef: Message [0..1]	A Message for the messageRef attribute MAY be entered. This indicates that the Message will be sent by the Task. The Message in this context is equivalent to an *out-only* message pattern (Web service). One or more corresponding outgoing Message Flow MAY be shown on the diagram. However, the display of the Message Flow is not required. The Message is applied to all outgoing Message Flow and the Message will be sent down all outgoing Message Flow at the completion of a single instance of the Task.
operationRef: Operation	This attribute specifies the operation that is invoked by the Service Task.
implementation: Implementation = Web Service {Web Service | Other | Unspecified}	This attribute specifies the technology that will be used to send and receive the Messages. A Web service is the default technology.

Receive Task

A Receive Task is a simple Task that is designed to wait for a Message to arrive from an external Participant (relative to the Process). Once the Message has been received, the Task is completed.

The actual *Participant* from which the Message is received can be identified by connecting the Receive Task to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.

A Receive Task is often used to start a Process. In a sense, the Process is bootstrapped by the receipt of the Message. In order for the Task to Instantiate the Process it must meet one of the following conditions:

- The Process does not have a Start Event and the Receive Task has no *incoming* Sequence Flow.
- The *incoming* Sequence Flow for the Receive Task has a source of a Start Event.
    - Note that no other *incoming* Sequence Flow are allowed for that Receive Task (in particular, a *loop* connection from a downstream object).

A Receive Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is an unfilled envelope marker (the same marker as a *catch* Message Event) in the upper left corner of the shape that indicates that the Task is a Receive Task (see Figure 10-14).

- A Receive Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes an unfilled envelope marker that distinguishes the shape from other Task types (as shown in Figure 10-14).

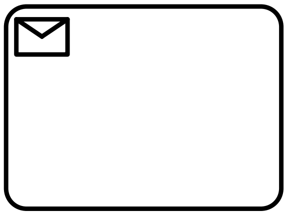


Figure 10-14 – A Receive Task Object

The Receive Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-10 presents the additional attributes and model associations of the Receive Task:

Table 10-10 – Receive Task attributes and model associations

Attribute Name	Description/Usage
messageRef: Message [0..1]	A Message for the messageRef attribute MAY be entered. This indicates that the Message will be received by the Task. The Message in this context is equivalent to an *in-only* message pattern (Web service). One or more corresponding incoming Message Flow MAY be shown on the diagram. However, the display of the Message Flow is not required. The Message is applied to all incoming Message Flow, but can arrive for only one of the incoming Message Flow for a single instance of the Task.
Instantiate: boolean = False	Receive Tasks can be defined as the instantiation mechanism for the Process with the Instantiate attribute. This attribute MAY be set to true if the Task is the first activity after the Start Event or a starting Task if there is no Start Event (i.e., there are no incoming Sequence Flow). Multiple Tasks MAY have this attribute set to True.
operationRef: Operation	This attribute specifies the operation that is invoked by the Service Task.
implementation: Implementation = Web Service {Web Service | Other | Unspecified}	This attribute specifies the technology that will be used to send and receive the Messages. A Web service is the default technology.

User Task

A User Task is a typical "workflow" Task where a human performer performs the Task with the assistance of a software application and is scheduled through a task list manager of some sort.

- A User Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a human figure marker that distinguishes the shape from other Task types (as shown in Figure 10-15).

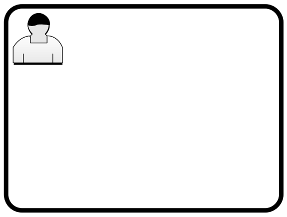


Figure 10-15 – A User Task Object

See “User Task” on page 177 within the larger section of “Human Interactions” for the details of User Tasks.

Manual Task

A Manual Task is a Task that is expected to be performed without the aid of any business process execution engine or any application. An example of this could be a telephone technician installing a telephone at a customer location.

- A Manual Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a hand figure marker that distinguishes the shape from other Task types (as shown in Figure 10-16).

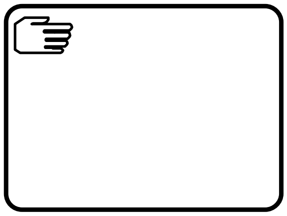


Figure 10-16 – A Manual Task Object

See “Manual Task” on page 176 within the larger section of “Human Interactions” for the details of Manual Tasks.

Business Rule

A Business Rule Task provides a mechanism for the Process to provide input to a Business Rules Engine and to get the output of calculations that the Business Rules Engine might provide. The InputOutputSpecification of the Task (see page 218) will allow the Process to send data to and receive data from the Business Rules Engine.

A Business Rule Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the Task is a Business Rule Task (see Figure 10-11).

- A Business Rule Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other Task types (as shown in Figure 10-17).

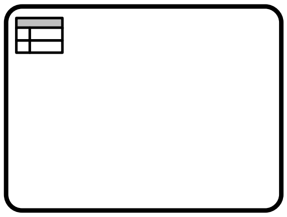


Figure 10-17 – A Business Rule Task Object

The Business Rule Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-11 presents the additional attributes of the Business Rule Task:

Table 10-11 – Business Rule Task attributes and model associations

Attribute Name	Description/Usage
Implementation: BusinesRuleTaskImplementation = Other {BuisnessRuleWebService | WebService | Other | Unspecified}	This attribute specifies the technology that will be used to implement the Business Rule Task

Script Task

A Script Task is executed by a business process engine. The modeler or implementer defines a script in a language that the engine can interpret. When the Task is ready to start, the engine will execute the script. When the script is completed, the Task will also be completed.

A Script Task object shares the same shape as the Task, which is a rectangle that has rounded corners. However, there is a graphical marker in the upper left corner of the shape that indicates that the Task is a Script Task (see Figure 10-11).

- A Script Task is a rounded corner rectangle that MUST be drawn with a single thin line and includes a marker that distinguishes the shape from other Task types (as shown in Figure 10-18).

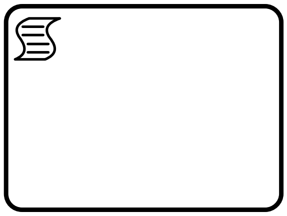


Figure 10-18 – A Script Task Object

The Script Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-12 presents the additional attributes of the Script Task:

Table 10-12 – Script Task attributes

Attribute Name	Description/Usage
scriptLanguage: string [0..1]	Defines the script language. The script language MUST be provided if a `script` is provided.
script: string [0..1]	The modeler MAY include a `script` that can be run when the Task is performed. If a script is not included, then the Task will act equivalent to a TaskType of None.

10.2.4. Human Interactions

Tasks with Human involvement

In many business workflows, human involvement is required to complete certain Tasks specified in the workflow model. BPMN specifies two different types of Tasks with human involvement, the Manual Task and the User Task.

A User Task is executed by and managed by a business process runtime. Attributes concerning the human involvement, like people assignments and UI rendering can be specified in great detail. A Manual Task is neither executed by nor managed by a business process runtime.

Notation

Both, the Manual Task and User Task share the same shape, which is a rectangle that has rounded corners. Manual Tasks and User Tasks have a Icons to indicate the human involvement required to complete the Task (see Figure 10-15 and Figure 10-16, above)

Manual Task

A Manual Task is a Task that is not managed by any business process engine. It can be considered as an unmanaged Task, unmanaged in the sense of that the business process engine doesn't track the start and completion of such a Task. An example of this could be a paper based instruction for a telephone technician to install a telephone at a customer location.

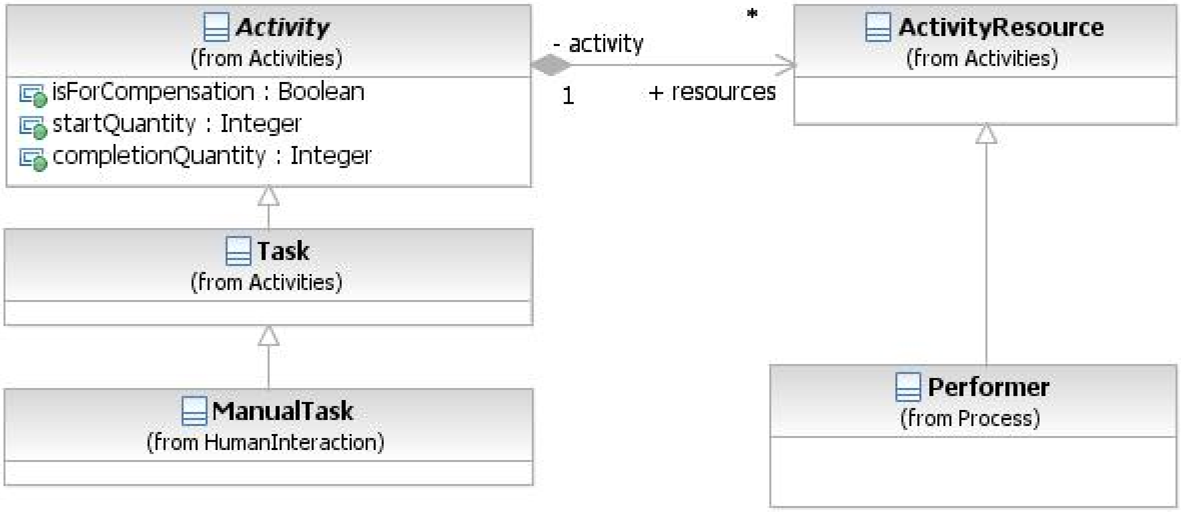


Figure 10-19 – Manual Task class diagram

The User Task inherits the attributes and model associations of Activity (see Table 10-3), but does not have any additional attributes or model associations.

User Task

A User Task is a typical “workflow” Task where a human performer performs the Task with the assistance of a software application, and where the Task is scheduled through a Task list manager of some sort.

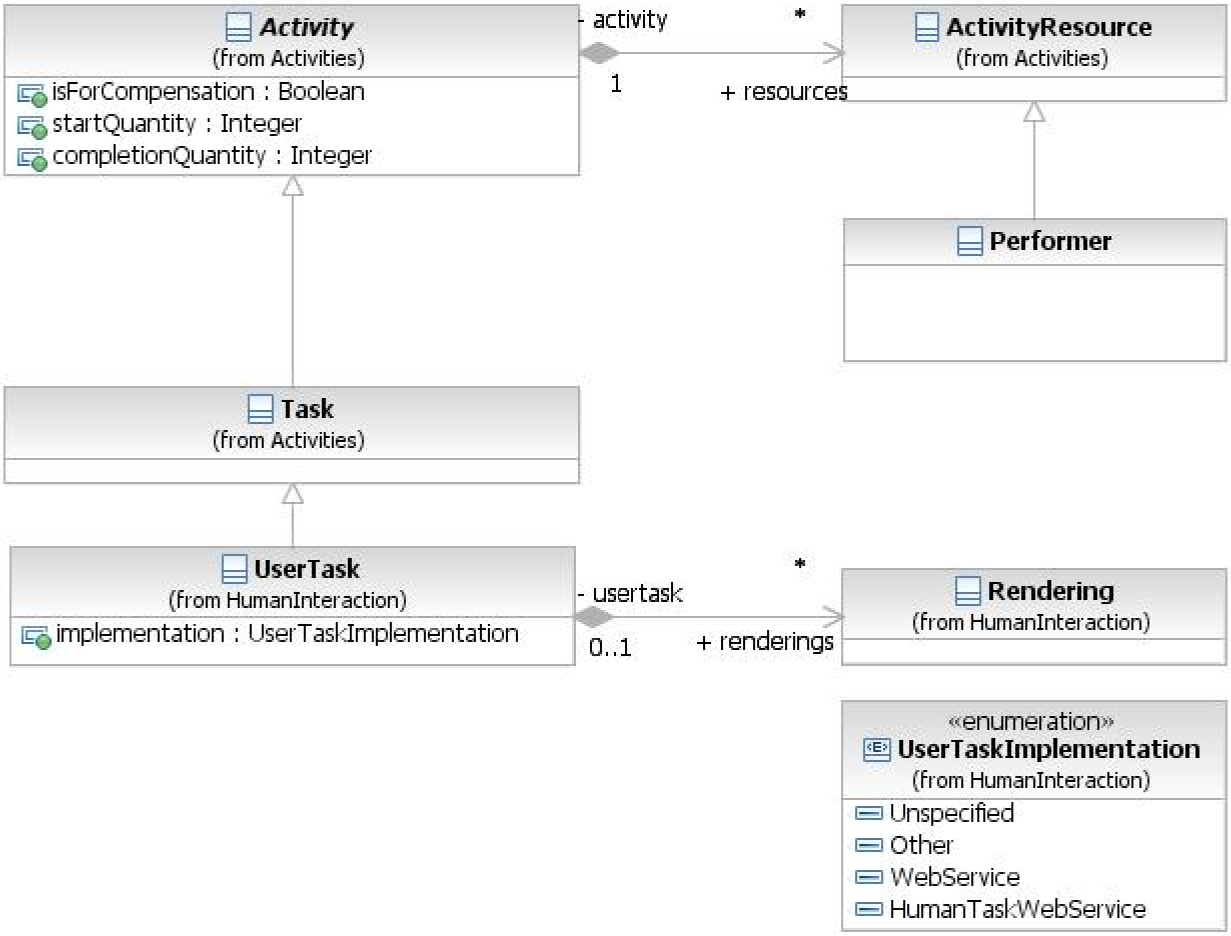


Figure 10-20 – User Task class diagram

The `User Task` can be implemented using different technologies, specified by the `UserTaskImplementation`.

The User Task inherits the attributes and model associations of Activity (see Table 10-3). Table 10-13 presents the additional attributes and model associations of the User Task:

Table 10-13 – User Task attributes and model associations

Attribute Name	Description/Usage
Implementation: UserTaskImplementation = Other {HumanTaskWebService | WebService | Other | Unspecified}	This attribute specifies the technology that will be used to implement the User Task
renderings: Rendering [0..*]	This attributes acts as a hook which allows BPMN adopters to specify task rendering attributes by using the BPMN Extension mechanism

The User Task inherits the Instance attributes of Activity (see Table 8-57). Table 10-14 presents the Instance attributes of the User Task element:

Table 10-14 – User Task *Instance* attributes

Attribute Name	Description/Usage
actualOwner: string	Returns the “user” who picked/claimed the User task and became the actual owner of it. The value is a literal representing the user’s id, email address etc.
taskPriority: integer	Returns the priority of the User Task

Rendering of User Tasks

BPMN User Tasks need to be rendered on user interfaces like forms clients, portlets, etc. The Rendering element provides an extensible mechanism for specifying UI renderings for User Tasks (Task UI). The element is optional. One or more rendering methods may be provided in a Task definition. A User Task can be deployed on any compliant implementation, irrespective of the fact whether the implementation supports specified rendering methods or not. The Rendering element is the extension point for renderings. Things like language considerations are opaque for the Rendering element because the rendering applications typically provide Multilanguage support. Where this is not the case, providers of certain rendering types may decide to extend the rendering type in order to provide language information for a given rendering. The content of the rendering element is not defined by this specification.

Human Performers

People can be assigned to Activities in various roles (called “generic human roles” in WS-HumanTask). BPMN 1.2 traditionally only has the *Performer* role. In addition to supporting the *Performer* role, BPMN 2.0

defines a specific HumanPerformer element allowing specifying more specific human roles as specialization of *HumanPerformer*, such as *PotentialOwner*.

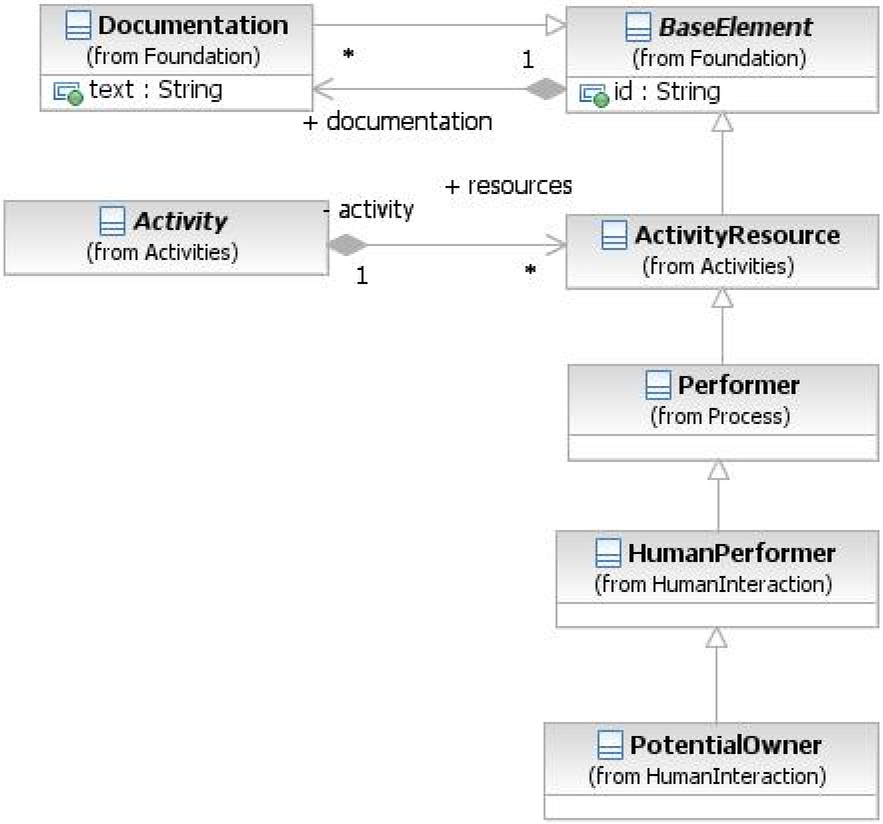


Figure 10-21 – HumanPerformer class diagram

The `HumanPerformer` element inherits the attributes and model associations of `ActivityResource` (see Table 10-5), through its relationship to `Performer`, but does not have any additional attributes or model associations.

Potential Owners

Potential owners of an `User Task` are persons who can claim and work on it. A potential owner becomes the actual owner of a `Task`, usually by explicitly claiming it.

XML Schema for Human Interactions

Table 10-15 – ManualTask XML schema

```
<xsd:element name="manualTask" type="tManualTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tManualTask">
        <xsd:complexContent>
                <xsd:extension base="tTask"/>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-16 – UserTask XML schema

```
<xsd:element name="userTask" type="tUserTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tUserTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:sequence>
                <xsd:element ref="rendering" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="implementation" type="tUserTaskImplementation" default="unspecified"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:element name="rendering" type="tRendering"/>
<xsd:complexType name="tRendering">
        <xsd:complexContent>
                <xsd:extension base="tBaseElement"/>
        </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tUserTaskImplementation">
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="unspecified"/>
                <xsd:enumeration value="other"/>
                <xsd:enumeration value="webService"/>
                <xsd:enumeration value="humanTaskWebService"/>
        </xsd:restriction>
</xsd:simpleType>
```

Table 10-17 – HumanPerformer XML schema

```
<xsd:element name="humanPerformer" type="tHumanPerformer" substitutionGroup="performer"/>
<xsd:complexType name="tHumanPerformer">
        <xsd:complexContent>
                <xsd:extension base="tPerformer">
                        <xsd:sequence>
                                <xsd:element ref="peopleAssignment" minOccurs="1" maxOccurs="1"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-18 – PotentialOwner XML schema

```
<xsd:element name="potentialOwner" type="tPotentialOwner" substitutionGroup="performer"/>
<xsd:complexType name="tPotentialOwner">
        <xsd:complexContent>
                <xsd:extension base="tHumanPerformer"/>
        </xsd:complexContent>
</xsd:complexType>
```

Examples

Consider the following sample procurement Process from the Buyer perspective

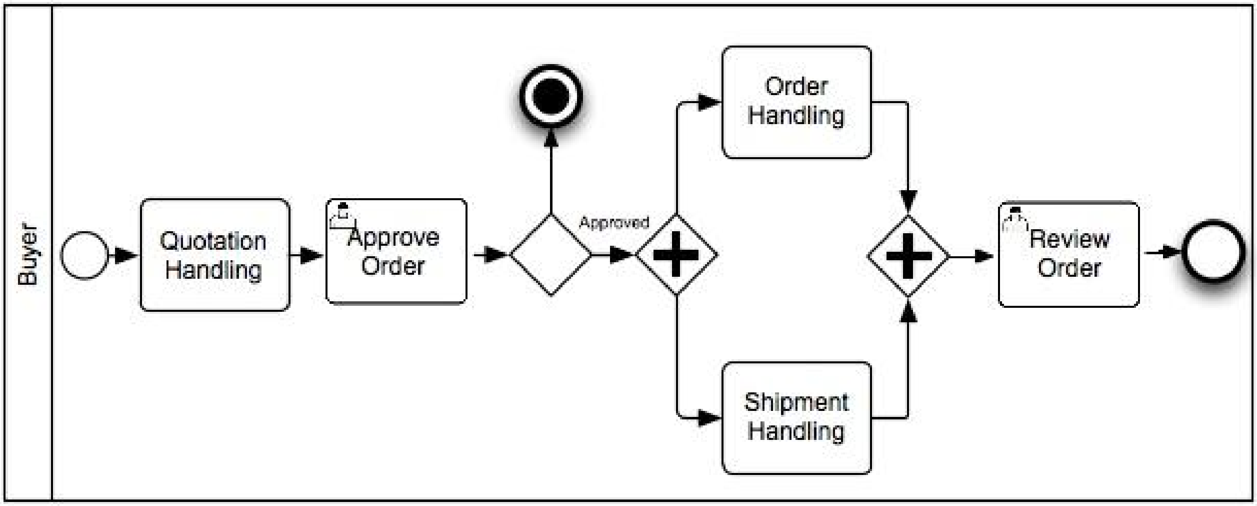


Figure 10-22 – Procurement Process Example

The Process comprises of two User Tasks

- **Approve Order**: After the quotation handling, the order needs to be approved by some regional manager to continue with the order and shipment handling
- **Review Order**: Once the order has been shipped to the Buyer, the order and shipment documents will be reviewed again by someone.

The details of the `Resource` and resource assignments are not shown in the BPMN above. See below XML sample of the "Buyer" `Process` for the `Resource` usage and resource assignments for potential owners.

Table 10-19 – XML serialization of Buyer process

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="Definition"
             targetNamespace="http://www.example.org/UserTaskExample"
             typeLanguage="http://www.w3.org/2001/XMLSchema"
             expressionLanguage="http://www.w3.org/1999/XPath"
             xmlns="http://www.omg.org/bpmn20"
             xmlns:tns="http://www.example.org/UserTaskExample">

  <resource id="regionalManager" name="Regional Manager">
    <resourceParameter id="buyerName" isRequired="true" name="Buyer Name" type="xsd:string"/>
    <resourceParameter id="region" isRequired="false" name="Region" type="xsd:string"/>
  </resource>

  <resource id="departmentalReviewer" name="Departmental Reviewer">
    <resourceParameter id="buyerName" isRequired="true" name="Buyer Name" type="xsd:string"/>
  </resource>

   <!-- Process definition -->
  <process id="Buyer" name="Buyer Process">
    <startEvent id="StartProcess"/>

    <sequenceFlow sourceRef="StartProcess" targetRef="QuotationHandling"/>

    <task id="QuotationHandling" name="Quotation Handling"/>

    <sequenceFlow sourceRef="QuotationHandling" targetRef="ApproveOrder"/>

    <userTask id="ApproveOrder" name="ApproveOrder">
      <potentialOwner resourceRef="tns:regionalManager">
        <resourceParameterBinding parameterRef="tns:buyerName">
          <formalExpression>getDataInput('order')/address/name</formalExpression>
        </resourceParameterBinding>
        <resourceParameterBinding parameterRef="tns:region">
          <formalExpression>getDataInput('order')/address/country</formalExpression>
        </resourceParameterBinding>
      </potentialOwner>
    </userTask>

    <sequenceFlow sourceRef="ApproveOrder" targetRef="OrderApprovedDecision"/>

    <exclusiveGateway id="OrderApprovedDecision" gatewayDirection="diverging"/>

    <sequenceFlow sourceRef="OrderApprovedDecision" targetRef="OrderAndShipment">
      <conditionExpression>Was the Order Approved?</conditionExpression>
    </sequenceFlow>
    <sequenceFlow sourceRef="OrderApprovedDecision" targetRef="TerminateProcess">
      <conditionExpression>Was the Order NOT Approved?</conditionExpression>
    </sequenceFlow>

    <endEvent id="TerminateProcess">
      <terminateEventDefinition id="TerminateEvent"/>
    </endEvent>

    <parallelGateway id="OrderAndShipment" gatewayDirection="diverging"/>

    <sequenceFlow sourceRef="OrderAndShipment" targetRef="OrderHandling"/>
    <sequenceFlow sourceRef="OrderAndShipment" targetRef="ShipmentHandling"/>

    <task id="OrderHandling" name="Order Handling"/>

    <task id="ShipmentHandling" name="Shipment Handling"/>

    <sequenceFlow sourceRef="OrderHandling" targetRef="OrderAndShipmentMerge"/>
    <sequenceFlow sourceRef="ShipmentHandling" targetRef="OrderAndShipmentMerge"/>

    <parallelGateway id="OrderAndShipmentMerge" gatewayDirection="converging"/>

    <userTask id="ReviewOrder" name="Review Order">
      <potentialOwner resourceRef="tns:departmentalReviewer">
        <resourceParameterBinding parameterRef="tns:buyerName">
          <formalExpression>getDataInput('order')/address/name</formalExpression>
        </resourceParameterBinding>
      </potentialOwner>
    </userTask>

    <sequenceFlow sourceRef="ReviewOrder" targetRef="EndProcess"/>

    <endEvent id="EndProcess"/>

  </process>
</definitions>
```

10.2.5. Sub-Processes

A Sub-Process is an Activity whose internal details have been modeled using Activities, Gateways, Events, and Sequence Flow. A Sub-Process is a graphical object within a Process, but it also can be "opened up" to show a lower-level Process. Sub-Processes define a contextual *scope* that can be used for attribute visibility, transactional *scope*, for the handling of *exceptions* (see page 283 for more details), of Events, or for *compensation* (see page 314 for more details).

There are different types of Sub-Processes, which will be described in the next five (5) sections.

Embedded Sub-Process (Sub-Process)

A Sub-Process object shares the same shape as the Task object, which is a rounded rectangle.

- A Sub-Process is a rounded corner rectangle that MUST be drawn with a single thin line.
  - The use of text, color, size, and lines for a Sub-Process MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63 with the exception that:
    - A boundary drawn with a thick line SHALL be reserved for Call Activity (Sub-Processes) (see page 196).
    - A boundary drawn with a dotted line SHALL be reserved for Event Sub-Processes (see page 188).
    - A boundary drawn with a double line SHALL be reserved for Transaction Sub-Processes (see page 188).

The Sub-Process can be in a collapsed view that hides its details (see Figure 10-23) or a Sub-Process can be in an expanded view that shows its details within the view of the Process in which it is contained (see Figure 10-24). In the collapsed form, the Sub-Process object uses a marker to distinguish it as a Sub-Process, rather than a Task.

- The Sub-Process marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.

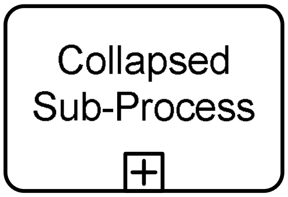


Figure 10-23 – A Sub-Process object (collapsed)

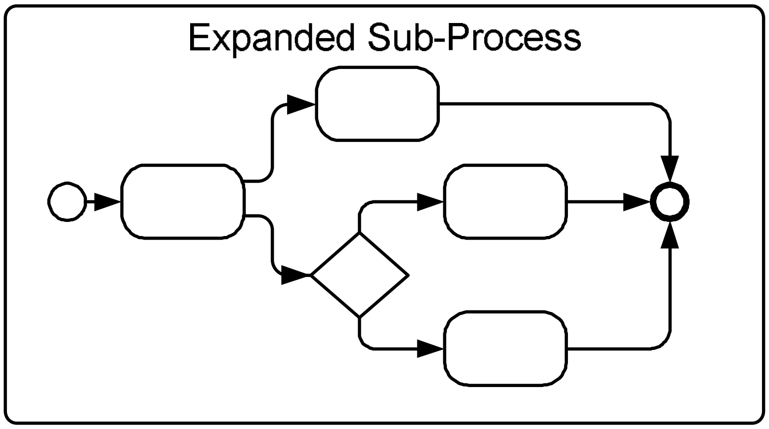


Figure 10-24 – A Sub-Process object (expanded)

They are used to create a context for exception handling that applies to a group of Activities (see page 283 for more details). *Compensations* can be handled similarly (see page 314 for more details).

Expanded Sub-Processes may be used as a mechanism for showing a group of parallel Activities in a less-cluttered, more compact way. In Figure 10-25, Activities “C” and “D” are enclosed in an unlabeled expanded Sub-Process. These two Activities will be performed in parallel. Notice that the expanded Sub-Process does not include a Start Event or an End Event and the Sequence Flow to/from these

Events. This usage of expanded Sub-Processes for "parallel boxes" is the motivation for having Start and End Events being optional objects.

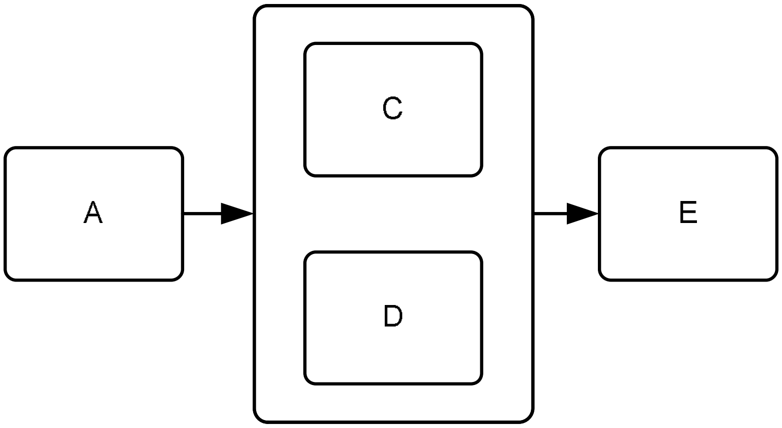


Figure 10-25 – Expanded Sub-Process used as a "Parallel Box"

BPMN specifies five (5) types of standard markers for Sub-Processes. The (Collapsed) Sub-Process marker, seen in Figure 10-23, can be combined with four (4) other markers: a *loop* marker or a *multi-instance* marker, a Compensation marker, and an Ad-Hoc marker. A collapsed Sub-Process may have one to three of these other markers, in all combinations except that *loop* and *multi-instance* cannot be shown at the same time (see Figure 10-26).

- The marker for a Sub-Process that *loops* MUST be a small line with an arrowhead that curls back upon itself.
  - The *loop* marker MAY be used in combination with any of the other markers except the *multi-instance* marker.
- The marker for a Sub-Process that has *Multiple Instances* MUST be a set of three vertical lines in parallel.
  - The *multi-instance* marker MAY be used in combination with any of the other markers except the *loop* marker.
- The marker for an *ad-hoc* Sub-Process MUST be a "tilde" symbol.
  - The *ad-hoc* marker MAY be used in combination with any of the other markers.
- The marker for a Sub-Process that is used for *compensation* MUST be a pair of left facing triangles (like a tape player "rewind" button).
  - The Compensation marker MAY be used in combination with any of the other markers.
- All the markers that are present MUST be grouped and the whole group centered at the bottom of the Sub-Process.

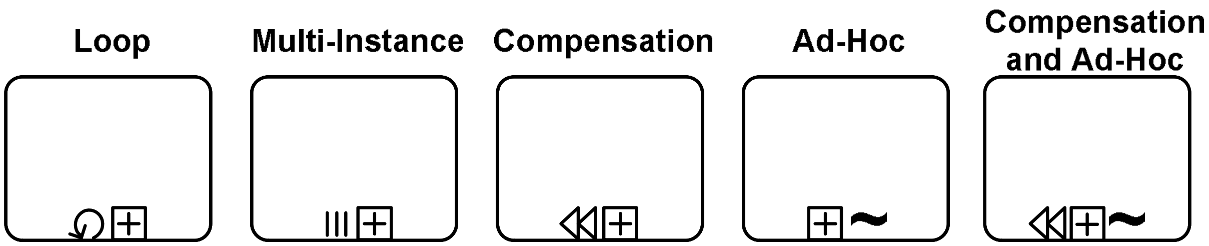


Figure 10-26 – Collapsed Sub-Process Markers

The Sub-Process now corresponds to the Embedded Sub-Process of BPMN 1.2. The Reusable Sub-Process of BPMN 1.2 corresponds to the Call Activity (calling a Process – see page 196).

Figure 10-27 shows the class diagram related to Sub-Processes.

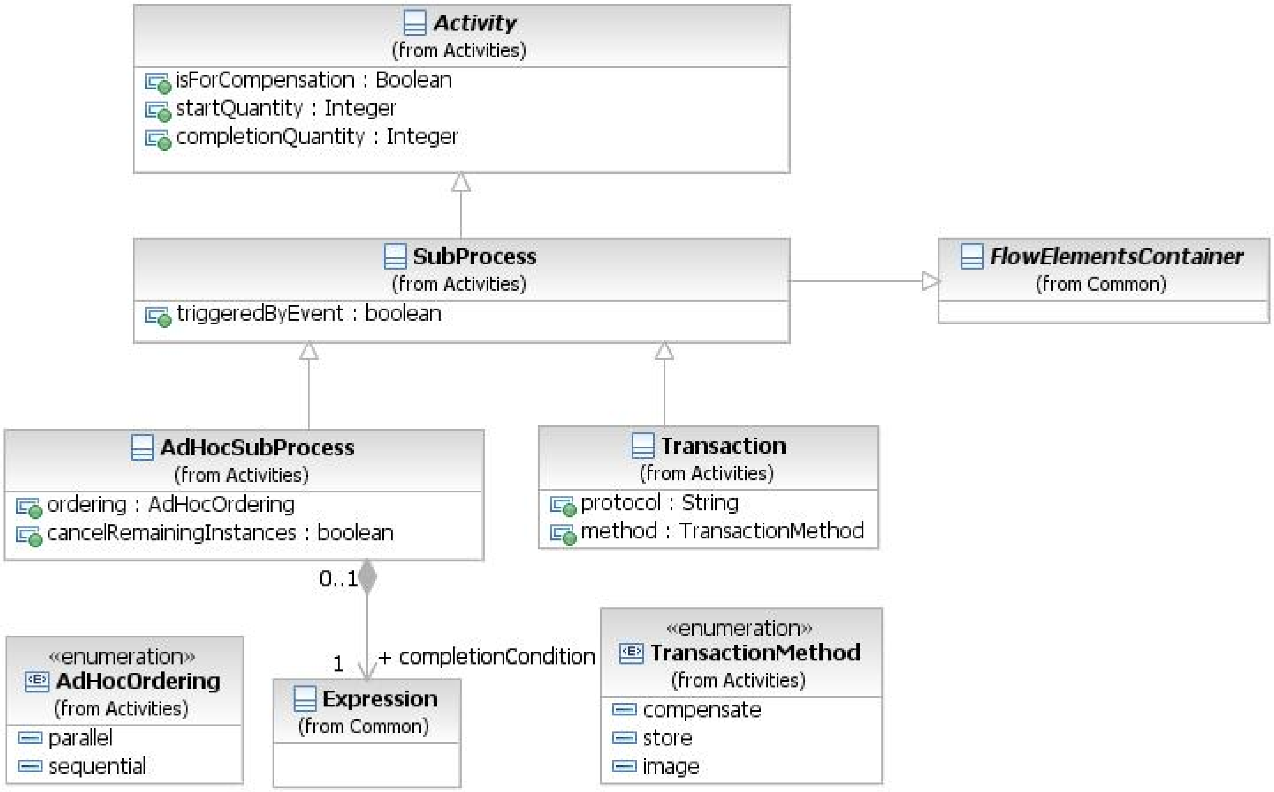


Figure 10-27 – The Sub-Process class diagram

The Sub-Process element inherits the attributes and model associations of Activity (see Table 10-3) and of FlowElementContainer (see Table 8-46). Table 10-3 presents the additional attributes of the Sub-Process element:

Table 10-20 – Sub-Process attributes

Attribute Name	Description/Usage
triggeredByEvent: boolean = false	A flag that identifies whether this Sub-Process is an Event Sub-Process. If *false*, then this Sub-Process is a normal Sub-Process. If *true*, the this Sub-Process is an Event Sub-Process and is subject to additional constraints (see page 188).

Reusable Sub-Process (Call Activity)

The *reusable* Sub-Process of BPMN 1.2 corresponds to the Call Activity that calls a pre-defined Process. See details of a Call Activity on page 196.

Event Sub-Process

An Event Sub-Process is a specialized Sub-Process that used within a Process (or Sub-Process). A Sub-Process is defined as an Event Sub-Process when its triggeredByEvent attribute is set to *true*.

An Event Sub-Process is not part of the *normal flow* of its parent Process—there are no *incoming* or *outgoing* Sequence Flow.

- An Event Sub-Process MUST NOT have any *incoming* or *outgoing* Sequence Flow.

An Event Sub-Process may or may not occur while the parent Process is active, but it is possible that it will occur many times. Unlike a standard Sub-Process, which uses the flow of the parent Process as a *trigger*, an Event Sub-Process has a Start Event with a *trigger*. Each time the Start Event is triggered while the parent Process is active, then the Event Sub-Process will start.

- The Start Event of an Event Sub-Process MUST have a defined *trigger*.
  - The Start Event *trigger* (EventDefinition) MUST be from the following types: Message, Error, Escalation, Compensation, Conditional, Signal, and Multiple (see page 266 for more details).
- An Event Sub-Process MUST have one and only one Start Event.

An Event Sub-Process object shares the same basic shape as the Sub-Process object, which is a rounded rectangle.

- An Event Sub-Process is a rounded corner rectangle that MUST be drawn with a single thin dotted line (see Figure 10-28 and Figure 10-29).
  - The use of text, color, size, and lines for a Sub-Process MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63 with the exception that:

- If the Event Sub-Process is collapsed, then its Start Event will be used as a marker in the upper left corner of the shape (see Figure 10-28).

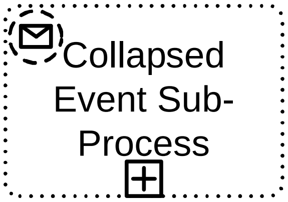


Figure 10-28 – An Event Sub-Process object (Collapsed)

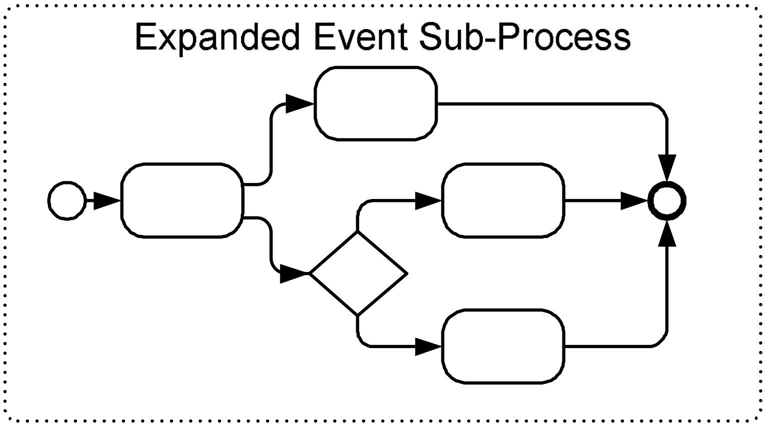


Figure 10-29 – An Event Sub-Process object (expanded)

There are two (2) possible consequences to the parent Process when an Event Sub-Process is triggered: 1) the parent Process can be interrupted, and 2) the parent Process can continue its work (not interrupted). This is determined by the type of Start Event that is used. See page 247 for the list of interrupting and non-interrupting Event Sub-Process Start Events.

Figure 10-30 provides an example of a Sub-Process that includes three (3) Event Sub-Processes. The first Event Sub-Process is triggered by a Message, does not interrupt the Sub-Process, and can occur multiple times. The second Event Sub-Process is used for *compensation* and will only occur after the Sub-Process has completed. The third Event Sub-Process handles errors that occur while the Sub-Process is active and will stop (interrupt) the Sub-Process if triggered.

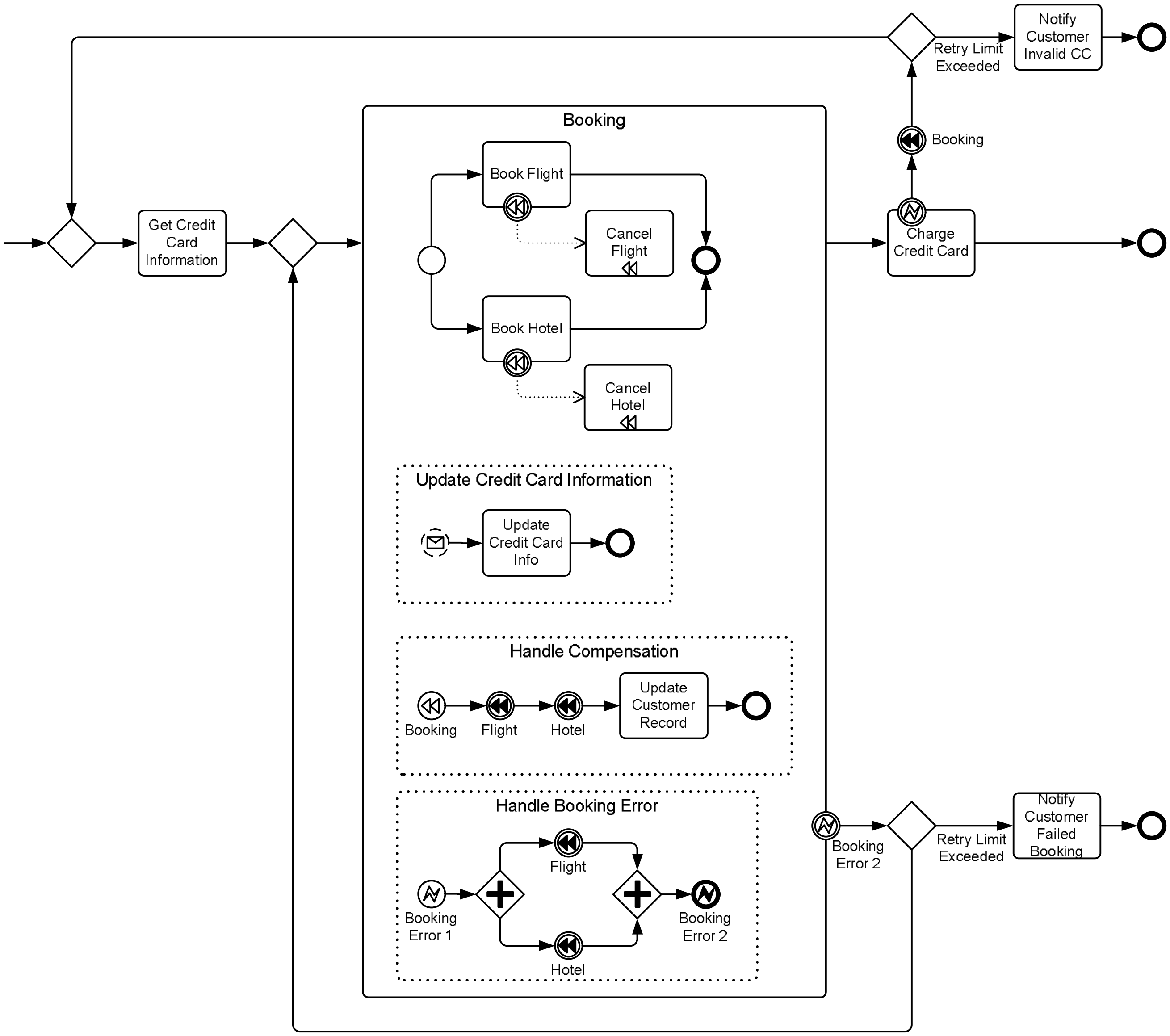


Figure 10-30 – An example that includes Event Sub-Processes

Transaction

A Transaction is a specialized type of Sub-Process which will have a special behavior that is controlled through a transaction protocol (such as WS-Transaction). The boundary of the Sub-Process will be double-lined to indicate that it is a Transaction (see Figure 10-31).

- A Transaction Sub-Process is a rounded corner rectangle that MUST be drawn with a double thin line.
  - The use of text, color, size, and lines for a *transaction* Sub-Process MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.

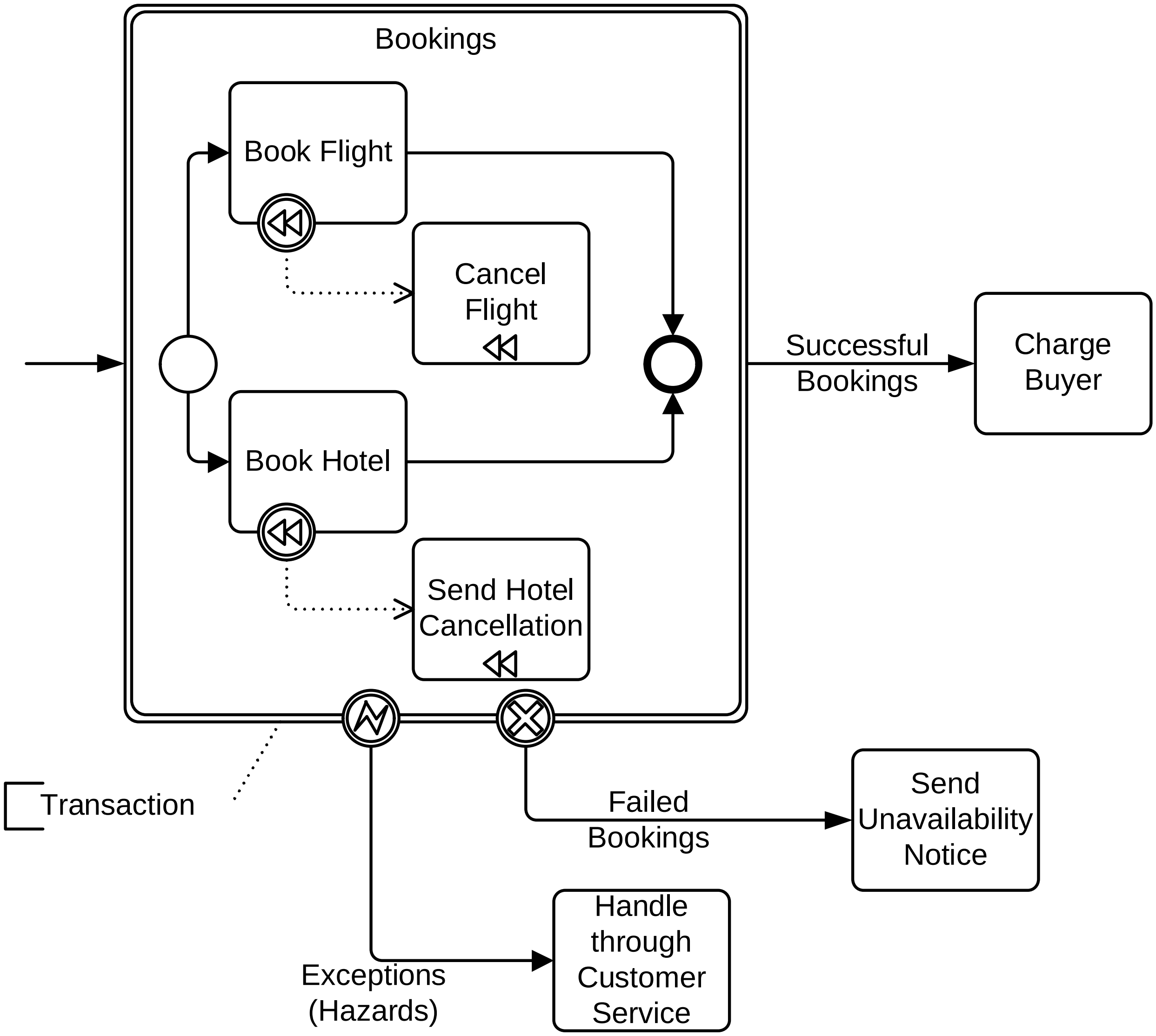


Figure 10-31 – A Transaction Sub-Process

The Transaction Sub-Process element inherits the attributes and model associations of Activities (see Table 10-3) through its relationship to Sub-Process. Table 10-21 presents the additional attributes and model associations of the Transaction Sub-Process:

Table 10-21 – Transaction Sub-Process attributes and model associations

Attribute Name	Description/Usage
protocol: string [0..1]	The elements that make up the internal Sub-Process flow. This association is only applicable when the XSD Interchange is used. In the case of the XMI interchange, this association is inherited from the FlowElementsContainer class.
method: TransactionMethod = compensate { compensate | store | image }	TransactionMethod is an attribute that defines the technique that will be used to undo a Transaction that has been cancelled. The default is compensate, but the attribute MAY be set to store or IMAGE.

There are three basic outcomes of a Transaction:

- **Successful completion**: this will be shown as a normal Sequence Flow that leaves the Transaction Sub-Process.
- **Failed completion (Cancel)**: When a Transaction is cancelled, the Activities inside the Transaction will be subjected to the cancellation actions, which could include rolling back the Process and *compensation* (see page 314 for more information on *compensation*) for specific Activities. Note that other mechanisms for interrupting a Transaction Sub-Process will not cause *compensation* (e.g., Error, Timer, and anything for a non-Transaction Activity). A Cancel Intermediate Event, attached to the boundary of the Activity, will direct the flow after the Transaction has been rolled back and all *compensation* has been completed. The Cancel Intermediate Event can only be used when attached to the boundary of a Transaction Sub-Process. It cannot be used in any *normal flow* and cannot be attached to a non-Transaction Sub-Process. There are two mechanisms that can signal the cancellation of a Transaction:
    - A Cancel End Event is reached within the *transaction* Sub-Process. A Cancel End Event can only be used within a *transaction* Sub-Process.
    - A *cancel* Message can be received via the transaction protocol that is supporting the execution of the Transaction Sub-Process.
- **Hazard**: This means that something went terribly wrong and that a normal success or cancel is not possible. Error Intermediate Events are used to show *Hazards*. When a *Hazard* happens, the Activity is interrupted (without *compensation*) and the flow will continue from the Error Intermediate Event.

The behavior at the end of a successful Transaction Sub-Process is slightly different than that of a normal Sub-Process. When each path of the Transaction Sub-Process reaches a non-Cancel End Event(s), the flow does not immediately move back up to the higher-level *parent* Process, as does a normal Sub-Process. First, the transaction protocol must verify that all the *Participants* have successfully completed their end of the Transaction. Most of the time this will be true and the flow will then move up to the higher-level Process. But it is possible that one of the *Participants* may end up with a problem that causes a *Cancel* or a *Hazard*. In this case, the flow will then move to the appropriate Intermediate Event, even though it had apparently finished successfully.

Ad-Hoc Sub-Process

An Ad-Hoc Sub-Process is a specialized type of Sub-Process that is a group of Activities that have no required sequence relationships. A set of activities can be defined for the Process, but the sequence and number of performances for the Activities is determined by the performers of the Activities.

A Sub-Process is marked as being *ad-hoc* with a "tilde" symbol placed at the bottom center of the Sub-Process shape (see Figure 10-32 and Figure 10-33).

- The marker for an Ad-Hoc Sub-Process MUST be a "tilde" symbol.
  - The Ad-Hoc Marker MAY be used in combination with any of the other markers.

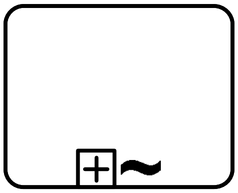


Figure 10-32 – A collapsed Ad-Hoc Sub-Process

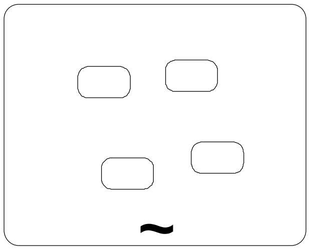


Figure 10-33 – An expanded Ad-Hoc Sub-Process

The Ad-Hoc Sub-Process element inherits the attributes and model associations of Activities (see Table 10-3) through its relationship to Sub-Process. Table 10-22 presents the additional model associations of the Ad-Hoc Sub-Process:

Table 10-22 – Ad-hoc Sub-Process model associations

Attribute Name	Description/Usage
completionCondition: Expression	This Expression defines the conditions when the Process will end. When the Expression is evaluated to *True*, the Process will be terminated.
ordering: AdHocOrdering = parallel { parallel | sequential }	This attribute defines if the Activities within the Process can be performed in parallel or must be performed sequentially. The default setting is parallel and the setting of sequential is a restriction on the performance that may be required due to shared resources. When the setting is sequential, then only one Activity can be performed at a time. When the setting is parallel, then zero (0) to all the Activities of the Sub-Process can be performed in parallel.
cancelRemainingInstances: Boolean = True	This attribute is used only if ordering is parallel. It determines whether running instances are cancelled when the completionCondition becomes *true*.

Activities within the Process are generally disconnected from each other. During execution of the Process, any one or more of the Activities may be active and they may be performed multiple times. The *performers* determine when Activities will start, what the next Activity will be, and so on.

Examples of the types of Processes that are Ad-Hoc include computer code development (at a low level), sales support, and writing a book chapter. If we look at the details of writing a book chapter, we could see that the Activities within this Process include: researching the topic, writing text, editing text, generating graphics, including graphics in the text, organizing references, etc. (see Figure 10-34). There may be some dependencies between Tasks in this Process, such as writing text before editing text, but there is not necessarily any correlation between an *instance* of writing text to an *instance* of editing text. Editing may occur infrequently and based on the text of many *instances* of the writing text Task.

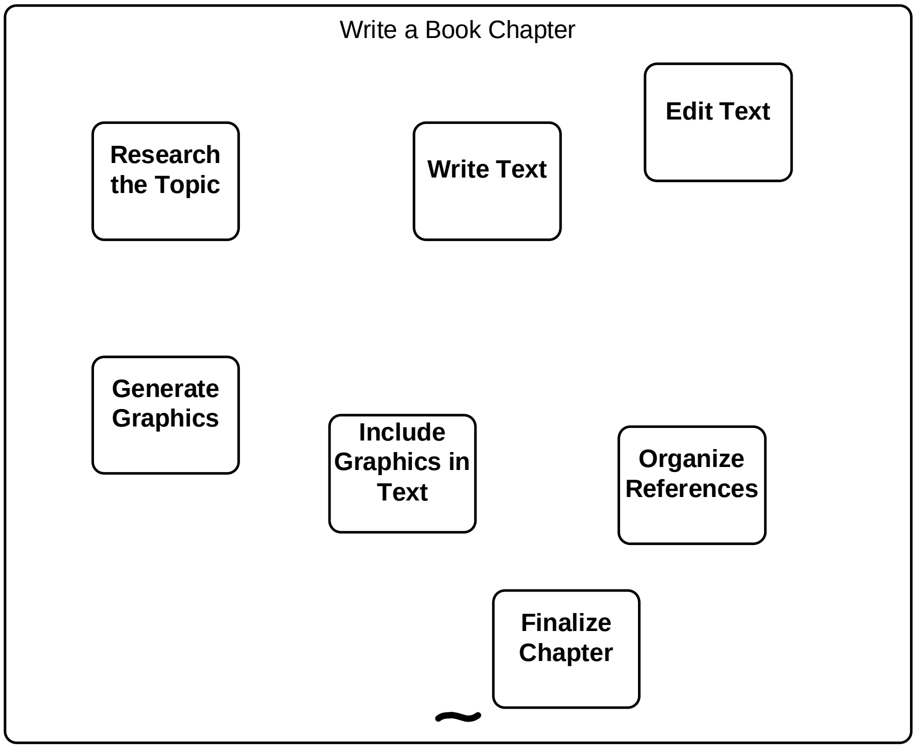


Figure 10-34 – An Ad-Hoc Sub-Process for writing a book chapter

Although there is no required formal Process structure, some sequence and data dependencies can be added to the details of the Process. For example, we can extend the book chapter Ad-Hoc Sub-Process shown above and add Data Objects, Data Associations, and even Sequence Flow (Figure 10-35).

Ad-Hoc Sub-Processes restrict the use of BPMN elements that would normally be used in Sub-Processes.

- The list of BPMN elements that MUST be used in an Ad-Hoc Sub-Process: Activity.
- The list of BPMN elements that MAY be used in an Ad-Hoc Sub-Process: Data Object, Sequence Flow, Association, Data Association, Group, Message Flow (as a *source* or *target*), Gateway, and Intermediate Event.
- The list of BPMN elements that MUST NOT be used in an Ad-Hoc Sub-Process: Start Event, End Event, Conversations (graphically), Conversation Links, and Choreography Activities.

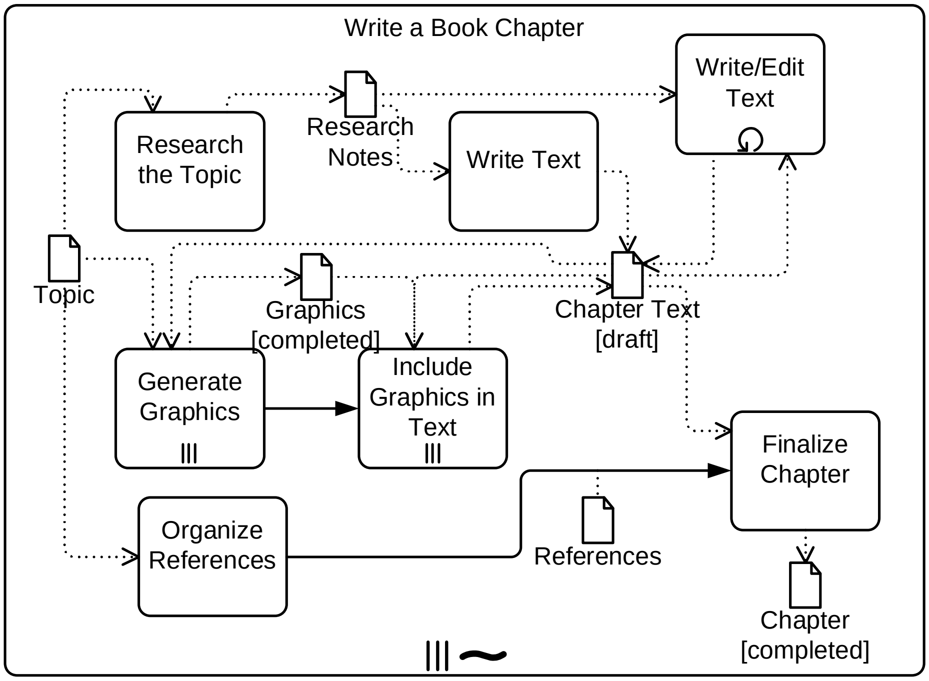


Figure 10-35 – An Ad-Hoc Sub-Process with data and sequence dependencies

The Data Objects as *inputs* into the Tasks act as an additional constraint for the performance of those Tasks. The *performers* still determine when the Tasks will be performed, but they are now constrained in that they cannot start the Task without the appropriate *input*. The addition of Sequence Flow between the Tasks (e.g., between "Generate Graphics" and "Include Graphics in Text") creates a dependency where the performance of the first Task must be followed by a performance of the second Task. This does not mean that the second Task must be performed immediately, but there must be a performance of the second Task after the performance of the first Task.

It is a challenge for a BPM engine to monitor the status of Ad-Hoc Sub-Processes, usually these kind of Processes are handled through groupware applications (such as e-mail), but BPMN allows modeling of Processes that are not necessarily executable, although there are some process engines that can follow an Ad-Hoc Sub-Process. Given this, at some point the Ad-Hoc Sub-Process will have complete and this can be determined by evaluating a completionCondition that evaluates Process attributes that will have been updated by an Activity in the Process.

10.2.6. Call Activity

A Call Activity identifies a point in the Process where a global Process or a Global Task is used. The Call Activity acts as a 'wrapper' for the invocation of a global Process or Global Task within the execution. The activation of a call Activity results in the transfer of control to the called global Process or Global Task.

The BPMN 2.0 Call Activity corresponds to the *Reusable* Sub-Process of BPMN 1.2. A BPMN 2.0 Sub-Process corresponds to the *Embedded* Sub-Process of BPMN 1.2 (see the previous section).

A Call Activity object shares the same shape as the Task and Sub-Process, which is a rectangle that has rounded corners. However, the target of what the Activity calls will determine the details of its shape.

- If the Call Activity calls a Global Task, then the shape will be the same as a Task, but the boundary of the shape will MUST have a thick line (see Figure 10-36)
    - The Call Activity MUST display the marker of the type of Global Task (e.g., the Call Activity would display the User Task marker if calling a Global User Task).
- If the Call Activity calls a Process, then there are two (2) options:
    - The details of the called Process can be hidden and the shape of the Call Activity will be the same as a *collpased* Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 10-37).
    - If the details of the called Process are available, then the shape of the Call Activity will be the same as a *expanded* Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 10-38).

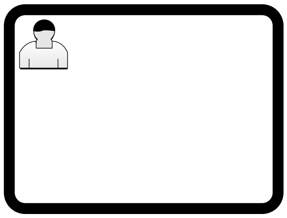


Figure 10-36 – A Call Activity object calling a Global Task

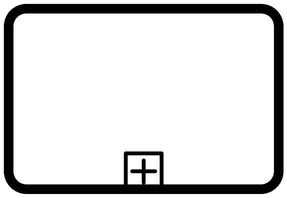


Figure 10-37 – A Call Activity object calling a Process (Collapsed)

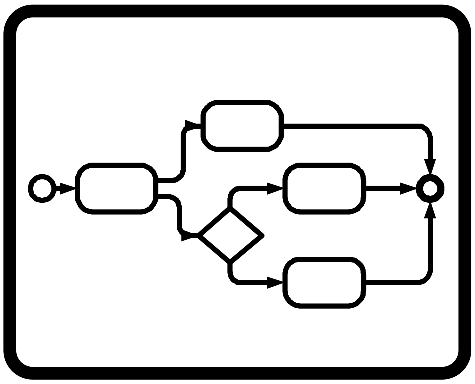


Figure 10-38 – A Call Activity object calling a Process (Expanded)

When a Process with a *definitional* Collaboration, calls a Process that also has a *definitional* Collaboration, the *Participants* of the two (2) Collaborations can be matched to each other using ParticipantAssociations of the Collaboration of the calling Process.

Since Call Activities rely in the CallableElement being invoked (see Figure 10-39), Call Activities must not define their own data *inputs*, InputSets, and *outputs*, OutputSets but use the data *inputs*, InputSets, and *outputs*, OutputSets defined in the referenced CallableElement

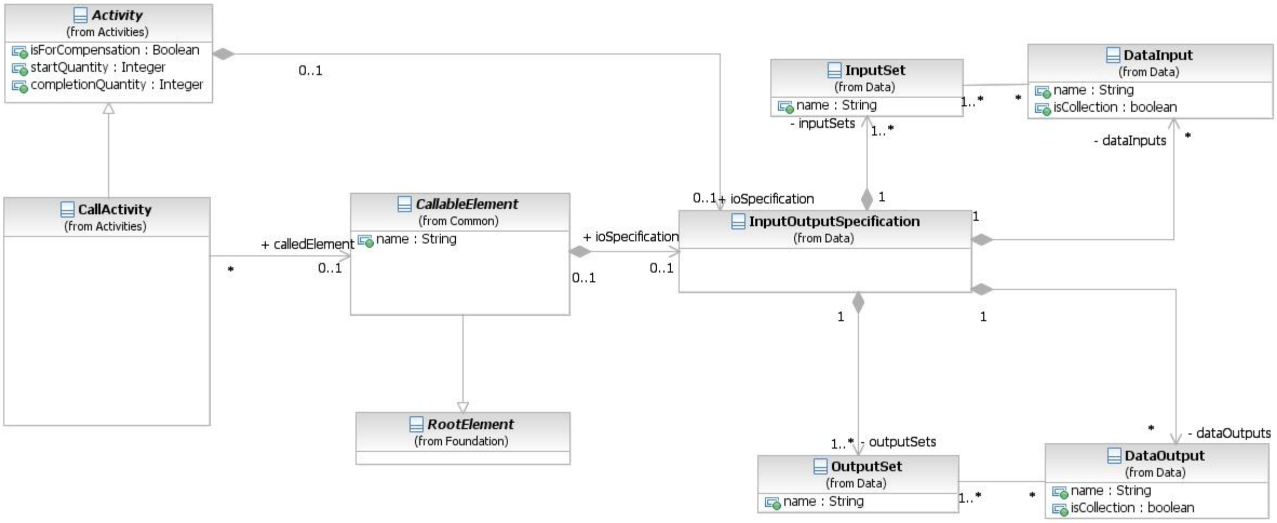


Figure 10-39 – The Call Activity class diagram

A Call Activity can override properties and attributes of the element being called, potentially changing the behavior of the called element based on the calling context. Also, Events that are propagated along the hierarchy (errors and escalations) are propagated from the called element to the Call Activity (and can be handled on its boundary).

The Call Activity inherits the attributes and model associations of Activity (see Table 10-3). Table 10-23 presents the additional model associations of the CallActivity:

Table 10-23 – CallActivity model associations

Attribute Name	Description/Usage
calledElement: CallableElement [0..1]	The element to be called, which will be either a Process or a GlobalTask. Other CallableElements, such as Choreography, GlobalChoreographyTask, Conversation, and GlobalCommunication MUST NOT be called by the Call Conversation element.

10.2.7. Global Task

A Global Task is a reusable, *atomic* Task definition that can be called from within any Process by a Call Activity.

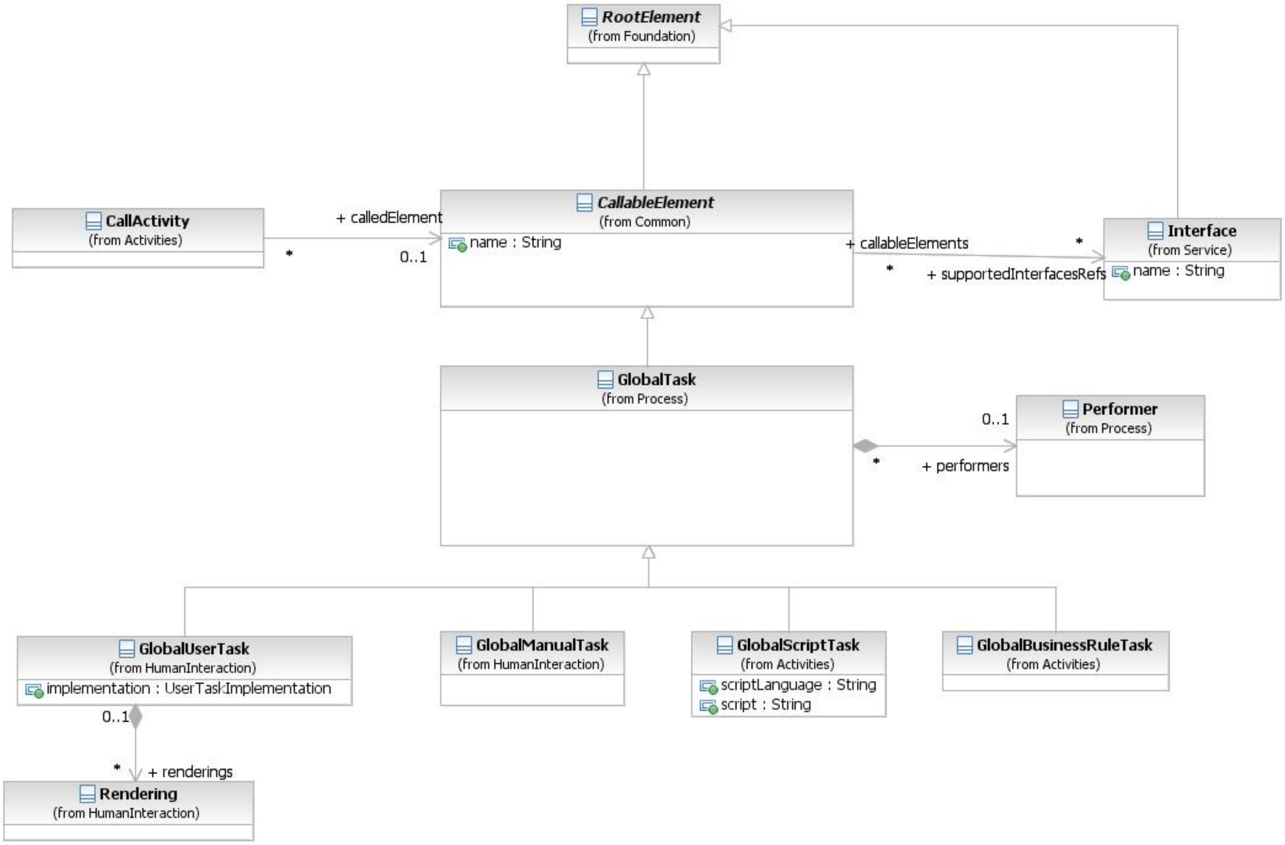


Figure 10-40 – Global Tasks class diagram

The Global Task inherits the attributes and model associations of Callable Element (see Table 8-30). There are no further attributes or model associations of the Global Task.

Types of Global Task

There are different types of Tasks identified within BPMN to separate the types of inherent behavior that Tasks might represent. This is true for both Global Tasks and standard Tasks, where the list of Task types is the same for both. For the sake of efficiency in this specification, the list of Task types is presented once on page 167. The behavior, attributes, and model associations defined in that section also apply to the types of Global Tasks.

10.2.8. Loop Characteristics

Activities may be repeated sequentially, essentially behaving like a *loop*. The presence of LoopCharacteristics signifies that the Activity has looping behavior. LoopCharacteristics is an abstract class. Concrete subclasses define specific kinds of looping behavior.

The LoopCharacteristics inherits the attributes and model associations of BaseElement (see Table 8-5). There are no further attributes or model associations of the LoopCharacteristics.

However, each Loop Activity *Instance* has attributes whose values may be referenced by Expressions. These values are only available when the Loop Activity is being executed.

Figure 10-41 displays the class diagram for an Activity's *loop* characteristics, including the details of both the standard *loop* and a *multi-instance*.

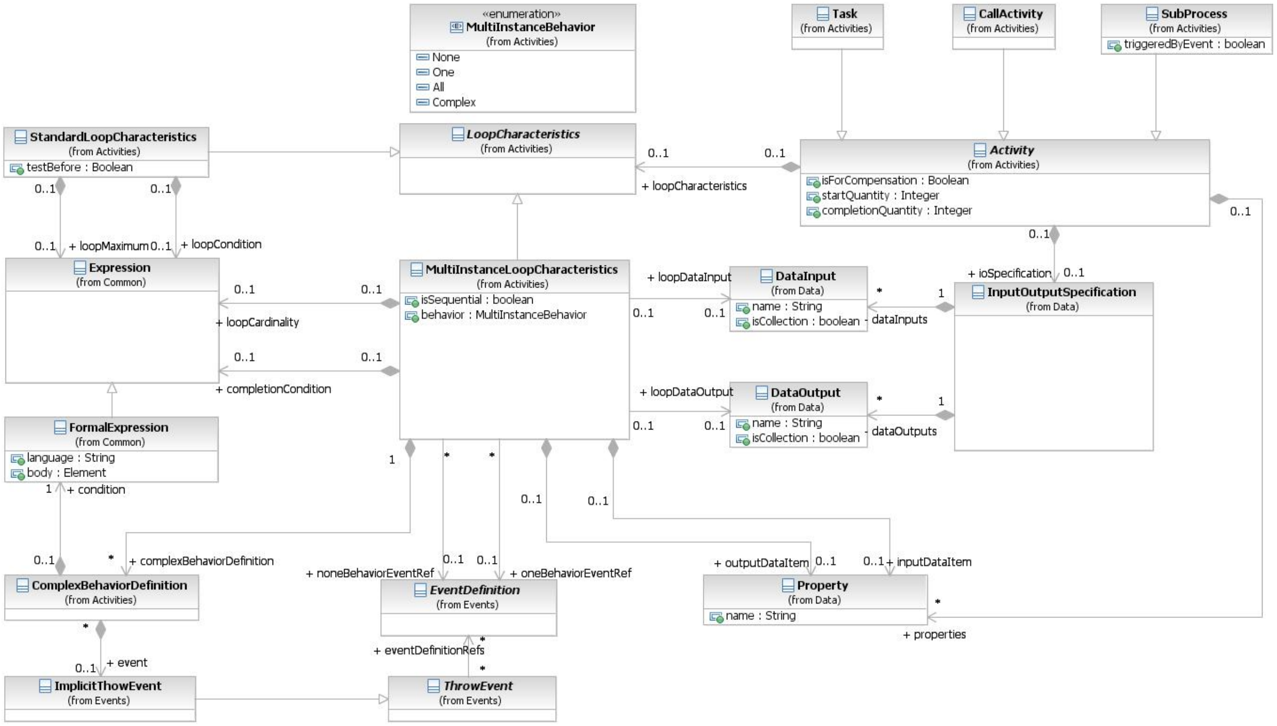


Figure 10-41 – LoopCharacteristics class diagram

The LoopCharacteristics element inherits the attributes and model associations of BaseElement (see Table 8-5), but does not have any further attributes or model associations. However, a Loop Activity does have additional *Instance* attributes as shown in Table 10-24.

Table 10-24 – Loop Activity *Instance* attributes

Attribute Name	Description/Usage
loopCounter: integer	The LoopCounter attribute is used at runtime to count the number of loops and is automatically updated by the process engine.

Standard Loop Characteristics

The StandardLoopCharacteristics class defines looping behavior based on a boolean condition. The Activity will *loop* as long as the boolean condition is *true*. The condition is evaluated for every *loop* iteration, and may be evaluated at the beginning or at the end of the iteration. In addition, a numeric cap can be optionally specified. The number of iterations may not exceed this cap.

- The marker for a Task or a Sub-Process that is a standard *loop* MUST be a small line with an arrowhead that curls back upon itself (see Figure 10-42 and Figure 10-43).

- The `loop` Marker MAY be used in combination with the `Compensation` Marker.

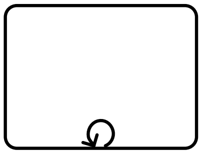


Figure 10-42 – A Task object with a Standard Loop Marker

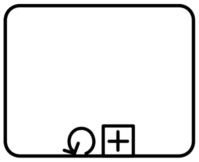


Figure 10-43 – A Sub-Process object with a Standard Loop Marker

The `StandardLoopCharacteristics` element inherits the attributes and model associations of `BaseElement` (see Table 8-5), through its relationship to `LoopCharacteristics`. Table 10-25 presents the additional attributes and model associations for the `StandardLoopCharacteristics` element:

Table 10-25 – StandardLoopCharacteristics attributes and model associations

Attribute Name	Description/Usage
testBefore: boolean = False	Flag that controls whether the loop condition is evaluated at the beginning (`testBefore` = *true*) or at the end (`testBefore` = *false*) of the loop iteration.
loopMaximum: integer [0..1]	Serves as a cap on the number of iterations.
loopCondition: Expression [0..1]	A Boolean expression that controls the loop. The `Activity` will only loop as long as this condition is *true*. The looping behavior may be underspecified, meaning that the modeler may simply document the condition, in which case the loop cannot be formally executed.

Multi-Instance Characteristics

The `MultiInstanceLoopCharacteristics` class allows for creation of a desired number of `Activity` *instances*. The *instances* may execute in parallel or may be sequential. Either an `expression` is used to specify or calculate the desired number of *instances* or a data driven setup can be used. In that case a data input can be specified, which is able to handle a collection of data. The number of items in the collection determines the number of `Activity` *instances*. This data input can be produced by a data input association. The modeler can also configure this *loop* to control the *Tokens* produced.

- The marker for a `Task` or `Sub-Process` that is a *multi-instance* MUST be a set of three vertical lines.
  - If the *multi-instance instances* are set to be performed in parallel rather than sequential (the

isSequential attribute set to *false*), then the lines of the marker will vertical (see Figure 10-44).

- If the *multi-instance instances* are set to be performed in sequence rather than parallel (the isSequential attribute set to *true*), then the marker will be horizontal (see Figure 10-45).
- The Multi-Instance marker MAY be used in combination with the Compensation marker.

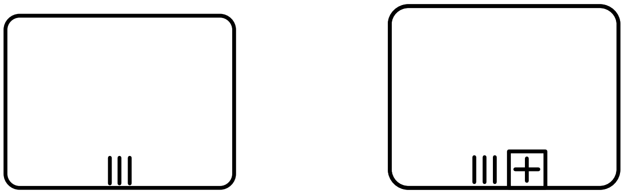


Figure 10-44 – Activity Multi-Instance marker for parallel *instances*

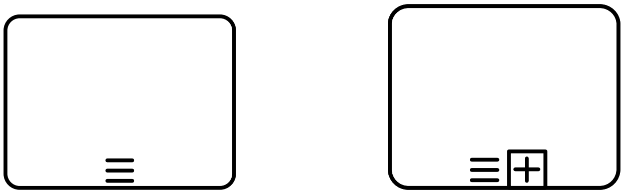


Figure 10-45 – Activity Multi-Instance marker for sequential *instances*

The MultiInstanceLoopCharacteristics element inherits the attributes and model associations of BaseElement (see Table 8-5), through its relationship to LoopCharacteristics. Table 10-26 presents the additional attributes and model associations for the MultiInstanceLoopCharacteristics element:

Table 10-26 – MultiInstanceLoopCharacteristics attributes and model associations

Attribute Name	Description/Usage
isSequential: boolean = False	This attribute is a flag that controls whether the Activity *instances* will execute sequentially or in parallel.
loopCardinality: Expression [0..1]	A numeric Expression that controls the number of Activity *instances* that will be created. This Expression MUST evaluate to an *integer*. This may be underspecified, meaning that the modeler may simply document the condition. In such a case the *loop* cannot be formally executed. In order to initialize a valid *multi-instance*, either the loopCardinality Expression or the loopDataInput MUST be specified.

loopDataInput: DataInput [0..1]	A reference to a `DataInput` which is part of the `Activity`’s `InputOutputSpecification`. This `DataInput` is used to determine the number of `Activity` *instances*, one `Activity` *instance* per item in the collection of data stored in that `DataInput` element. In order to initialize a valid *multi-instance*, either the `loopCardinality Expression` or the `loopDataInput` MUST be specified.
loopDataOutput: DataOutput [0..1]	A reference to a `DataOutput` which is part of the `Activity`’s `InputOutputSpecification`. This `DataOutput` specifies the collection of data, which will be produced by the *multi-instance*.
inputDataItem: Property [0..1]	A `Property`, representing for every `Activity` *instance* the single item of the collection stored in the `loopDataInput`. This `Property` can be the source of `DataInputAssociation` to a data input of the `Activity`’s `InputOutputSpecification`. The type of this `Property` MUST the scalar of the type defined for the `loopDataInput`.
outputDataItem: Property [0..1]	A `Property`, representing for every `Activity` *instance* the single item of the collection stored in the `loopDataOutput`. This `Property` can be the target of `DataOutputAssociation` to a data output of the `Activity`’s `InputOutputSpecification`. The type of this `Property` MUST the scalar of the type defined for the `loopDataOutput`.
behavior: MultiInstanceBehavior = all { none | one | all | complex }	The attribute behavior acts as a shortcut for specifying when events shall be thrown from an `Activity` *instance* that is about to complete. It can assume values of `none`, `one`, `all`, and `complex`, resulting in the following behavior: • **none**: the `EventDefinition` which is associated through the `noneEvent` association will be thrown for each *instance* completing; • **one**: the `EventDefinition` referenced through the `oneEvent` association will be thrown upon the first *instance* completing; • **`all`**: no `Event` is ever thrown; a *token* is produced after completion of all *instances* • **`complex`**: the `complexBehaviorDefinitions` are consulted to determine if and which `Events` to throw. For the behaviors of `none` and `one`, a default `SignalEventDefinition` will be thrown which automatically carries the current runtime attributes of the `MI Activity`. Any thrown `Events` can be caught by *boundary* `Events` on the `Multi-Instance Activity`.

complexBehaviorDefinition: ComplexBehaviorDefinition [0..*]	Controls when and which `Events` are thrown in case `behavior` is set to `complex`.
completionCondition: Expression [0..1]	This attribute defines a Boolean `Expression` that when evaluated to *true*, cancels the remaining `Activity` *instances* and produces a *token*.
oneBehaviorEventRef: EventDefinition [0..1]	The `EventDefinition` which is thrown when behavior is set to one and the first internal `Activity` *instance* has completed.
noneBehaviorEventRef: EventDefinition [0..1]	The `EventDefinition` which is thrown when the behavior is set to none and an internal `Activity` *instance* has completed.

The following table lists all *instance* attributes available at runtime. For each *instance* of the `Multi-Instance Activity` (outer *instance*), there exist a number of generated (inner) *instances* of the `Activity` at runtime.

Table 10-27 – Multi-instance Activity Instance attributes

Attribute Name	Description/Usage
loopCounter: integer	This attribute is provided for each generated (inner) *instance* of the `Activity`. It contains the sequence number of the generated *instance*, i.e., if this value of some *instance* in *n*, the *instance* is the *n*-th *instance* that was generated.
numberOfInstances: integer	This attribute is provided for the outer *instance* of the `Multi-Instance Activity` only. This attribute contains the total number of inner *instances* created for the `Multi-Instance Activity`.
numberOfActiveInstances: integer	This attribute is provided for the outer *instance* of the `Multi-Instance Activity` only. This attribute contains the number of currently active inner *instances* for the `Multi-Instance Activity`. In case of a sequential `Multi-Instance Activity`, this value can't be greater than 1. For parallel `Multi-Instance Activities`, this value can't be greater than the value contained in `numberOfInstances`
numberOfCompletedInstances: integer	This attribute is provided for the outer *instance* of the `Multi-Instance Activity` only. This attribute contains the number of already completed inner *instances* for the `Multi-Instance Activity`.
numberOfTerminatedInstances: integer	This attribute is provided for the outer *instance* of the `Multi-Instance Activity` only. This attribute contains the number of terminated inner *instances* for the `Multi-Instance Activity`. The sum of `numberOfTerminatedInstances`, `numberOfCompletedInstances` and `numberOfActiveInstances` always sums up to `numberOfInstances`.

Complex Behavior Definition

This element controls when and which Events are thrown in case behavior of the Multi-Instance Activity is set to complex.

The ComplexBehaviorDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-28 presents the additional attributes and model associations for the ComplexBehaviorDefinition element:

Table 10-28 – ComplexBehaviorDefinition attributes and model associations

Attribute Name	Description/Usage
condition: Formal Expression	This attribute defines a Boolean Expression that when evaluated to *true*, cancels the remaining Activity *instances* and produces a *token*.
event: ImplicitThrowEvent	If the condition is *true*, this identifies the Event that will be thrown (to be caught by a *boundary* Event on the Multi-Instance Activity)/

10.2.9. XML Schema for Activities

Table 10-29 – Activity XML schema

```
<xsd:element name="activity" type="tActivity"/>
<xsd:complexType name="tActivity" abstract="true">
      <xsd:complexContent>
        <xsd:extension base="tFlowNode">
            <xsd:sequence>
                <xsd:element ref="ioSpecification" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataInputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataOutputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="activityResource" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="loopCharacteristics" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="isForCompensation" type="xsd:boolean" default="false"/>
            <xsd:attribute name="default" type="xsd:IDREF" use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-30 – ActivityResource XML schema

```
<xsd:element name="activityResource" type="tActivityResource"/>
<xsd:complexType name="tActivityResource">
        <xsd:complexContent>
                <xsd:extension base="tBaseElement">
                        <xsd:sequence>
                                <xsd:element ref="resourceAssignmentExpression" minOccurs="0"
                                maxOccurs="1"/>
                                <xsd:element ref="resourceParameterBinding" minOccurs="0"
                                maxOccurs="unbounded"/>
                        </xsd:sequence>
                        <xsd:attribute name="resourceRef" type="xsd:QName" use="required"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-31 – AdHocSubProcess XML schema

```
<xsd:element name="adHocSubProcess" type="tAdHocSubProcess" substitutionGroup="flowElement"/>
<xsd:complexType name="tAdHocSubProcess">
    <xsd:complexContent>
        <xsd:extension base="tSubProcess">
            <xsd:sequence>
                <xsd:element name="completionCondition" type="tExpression" minOccurs="0"
                        maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="cancelRemainingInstances" type="xsd:boolean" default="true"/>
            <xsd:attribute name="ordering" type="tAdHocOrdering"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tAdHocOrdering">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="parallel"/>
        <xsd:enumeration value="sequential"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 10-32 – BusinessRuleTask XML schema

```
<xsd:element name="businessRuleTask" type="tBusinessRuleTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tBusinessRuleTask">
      <xsd:complexContent>
        <xsd:extension base="tTask"/>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-33 – CallActivity XML schema

```
<xsd:element name="callActivity" type="tCallActivity" substitutionGroup="flowElement"/>
<xsd:complexType name="tCallActivity">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:attribute name="calledElement" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-34 – GlobalBusinessRuleTask XML schema

```
<xsd:element name="globalBusinessRuleTask" type="tGlobalBusinessRuleTask"
        substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalBusinessRuleTask">
    <xsd:complexContent>
        <xsd:extension base="tGlobalTask"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-35 – GlobalScriptTask XML schema

```
<xsd:element name="globalScriptTask" type="tGlobalScriptTask"  substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalScriptTask">
      <xsd:complexContent>
        <xsd:extension base="tGlobalTask">
            <xsd:sequence>
                  <xsd:element ref="script" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="scriptLanguage" type="xsd:anyURI"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-36 – LoopCharacteristics XML schema

```
<xsd:element name="loopCharacteristics" type="tLoopCharacteristics"/>
<xsd:complexType name="tLoopCharacteristics" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-37 – MultiInstanceLoopCharacteristics XML schema

```
<xsd:element name="multiInstanceLoopCharacteristics" type="tMultiInstanceLoopCharacteristics"
        substitutionGroup="loopCharacteristics"/>
<xsd:complexType name="tMultiInstanceLoopCharacteristics">
    <xsd:complexContent>
    <xsd:extension base="tLoopCharacteristics">
        <xsd:sequence>
                <xsd:element name="loopCardinality" type="tExpression" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="loopDataInput" type="tDataInput" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="loopDataOutput" type="tDataOutput" minOccurs="0"
                        maxOccurs="1"/>
                <xsd:element name="inputDataItem" type="tProperty" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="outputDataItem" type="tProperty" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="complexBehaviorDefinition" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="completionCondition" type="tExpression" minOccurs="0"
                        maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="isSequential" type="xsd:boolean" default="false"/>
            <xsd:attribute name="behavior" type="tMultiInstanceFlowCondition" default="all"/>
            <xsd:attribute name="oneBehaviorEventRef" type="xsd:QName" use="optional"/>
            <xsd:attribute name="noneBehaviorEventRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tMultiInstanceFlowCondition">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="none"/>
        <xsd:enumeration value="one"/>
        <xsd:enumeration value="all"/>
        <xsd:enumeration value="complex"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 10-38 – ReceiveTask XML schema

```
<xsd:element name="receiveTask" type="tReceiveTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tReceiveTask">
    <xsd:complexContent>
        <xsd:extension base="tTask">
            <xsd:sequence>
                <xsd:element name="messageRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="operationRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="instantiate" type="xsd:boolean" default="false"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-39 – ScriptTask XML schema

```
<xsd:element name="scriptTask" type="tScriptTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tScriptTask">
       <xsd:complexContent>
          <xsd:extension base="tTask">
               <xsd:sequence>
                      <xsd:element ref="script" minOccurs="0" maxOccurs="1"/>
               </xsd:sequence>
               <xsd:attribute name="scriptLanguage" type="xsd:anyURI"/>
          </xsd:extension>
       </xsd:complexContent>
</xsd:complexType>

<xsd:element name="script" type="tScript"/>
<xsd:complexType name="tScript" mixed="true">
       <xsd:sequence>
          <xsd:any namespace="##any" processContents="lax" minOccurs="0"/>
       </xsd:sequence>
</xsd:complexType>
```

Table 10-40 – SendTask XML schema

```
<xsd:element name="sendTask" type="tSendTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tSendTask">
     <xsd:complexContent>
          <xsd:extension base="tTask">
               <xsd:sequence>
                    <xsd:element name="messageRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
                    <xsd:element name="operationRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
               </xsd:sequence>
          </xsd:extension>
     </xsd:complexContent>
</xsd:complexType>
```

Table 10-41 – ServiceTask XML schema

```
<xsd:element name="serviceTask" type="tServiceTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tServiceTask">
     <xsd:complexContent>
          <xsd:extension base="tTask">
               <xsd:sequence>
                    <xsd:element name="operationRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
                    <xsd:element name="serviceRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
               </xsd:sequence>
          </xsd:extension>
     </xsd:complexContent>
</xsd:complexType>
```

Table 10-42 – StandardLoopCharacteristics XML schema

```
<xsd:element name="standardLoopCharacteristics" type="tStandardLoopCharacteristics"/>
<xsd:complexType name="tStandardLoopCharacteristics">
    <xsd:complexContent>
        <xsd:extension base="tLoopCharacteristics">
            <xsd:sequence>
                <xsd:element name="loopCondition" type="tExpression" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="testBefore" type="xsd:boolean" default="false"/>
            <xsd:attribute name="loopMaximum" type="xsd:integer" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-43 – SubProcess XML schema

```
<xsd:element name="subProcess" type="tSubProcess" substitutionGroup="flowElement"/>
<xsd:complexType name="tSubProcess">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-44 – Task XML schema

```
<xsd:element name="task" type="tTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tTask">
    <xsd:complexContent>
        <xsd:extension base="tActivity"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-45 – Transaction XML schema

```
<xsd:element name="transaction" type="tTransaction" substitutionGroup="flowElement"/>
<xsd:complexType name="tTransaction">
    <xsd:complexContent>
        <xsd:extension base="tActivity"/>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tTransactionMethod">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="compensate"/>
        <xsd:enumeration value="image"/>
        <xsd:enumeration value="store"/>
    </xsd:restriction>
</xsd:simpleType>
```

10.3. Items and Data

A traditional requirement of Process modeling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a Process. An import aspect of this is the ability to capture the structure of that data and to query or manipulate that structure.

BPMN does not itself provide a built-in model for describing structure of data or an expression language for querying that data. Instead it formalizes hooks that allow for externally defined data structures and expression languages. In addition, BPMN allows for the co-existence of multiple data structure and expression languages within the same model. The compatibility and verification of these languages is outside the scope of this specification and becomes the responsibility of the tool vendor.

BPMN designates XML Schema and XPath as its default data structure and expression languages respectively, but vendors are free to substitute their own languages.

10.3.1. Data Modeling

A traditional requirement of Process modeling is to be able to model the items (physical or information items) that are created, manipulated, and used during the execution of a Process.

This requirement is realized in BPMN through various constructs: Data Objects, *ItemDefinition*, *Properties*, *Data Inputs*, *Data Outputs*, Messages, *Input Sets*, *Output Sets*, and Data Associations.

Item-Aware Elements

Several elements in BPMN are subject to store or convey items during process execution. These elements are referenced generally as “item-aware elements.” This is similar to the variable construct common to many languages. As with variables, these elements have a ItemDefinition.

The data structure these elements hold is specified using an associated ItemDefinition. An item-aware element may be underspecified, meaning that the structure attribute of its ItemDefinition is optional if the modeler does not wish to define the structure of the associated data.

The elements in the specification defined as item-aware elements are: Data Objects, Properties, DataInputs and DataOutputs, Messages.

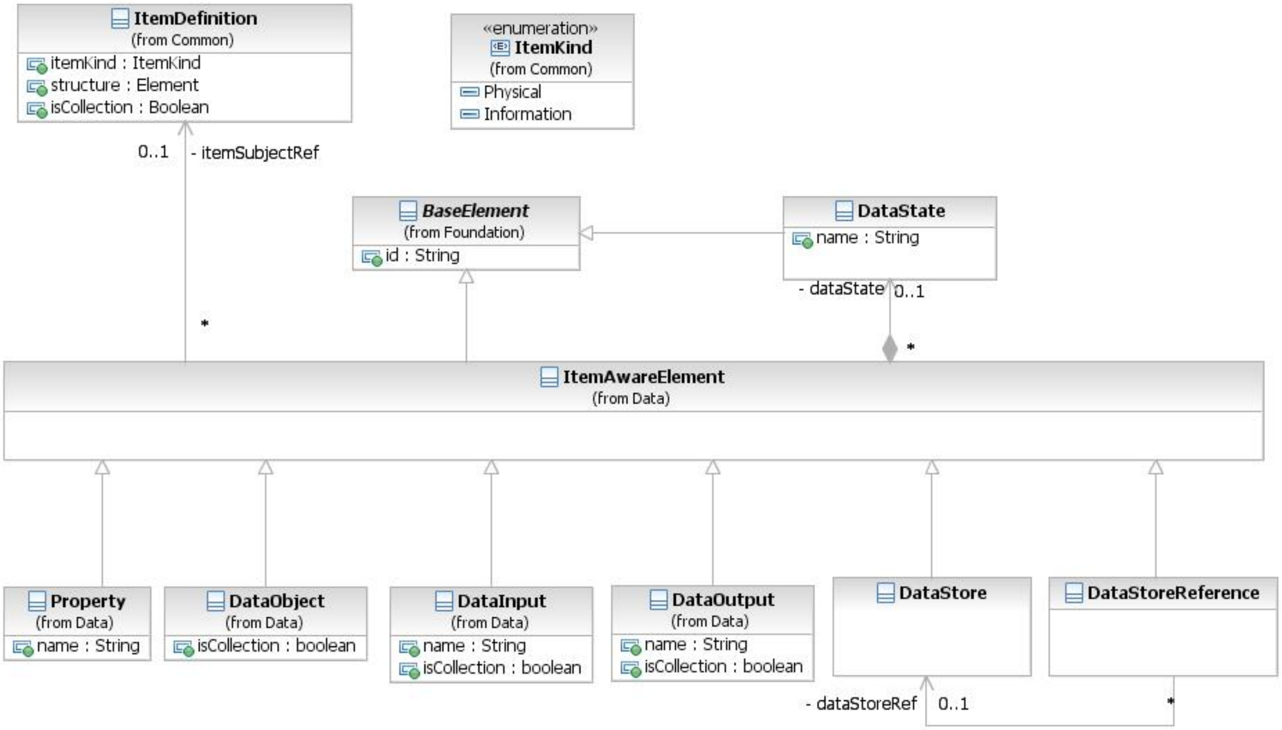


Figure 10-46 – ItemAware class diagram

The ItemAwareElement element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-46 presents the additional model associations of the ItemAwareElement element:

Table 10-46 – ItemAwareElement model associations

Attribute Name	Description/Usage
itemSubjectRef: ItemDefinition [0..1]	Specification of the items that are stored or conveyed by the ItemAwareElement.
dataState: DataState [0..1]	A reference to the DataState, which defines certain states for the data contained in the Item.

Data Objects

The primary construct for modeling data within the Process flow is the DataObject element. A DataObject has a well-defined lifecycle, with resulting visibility constraints.

DataObject

The DataObject class is an item-aware element. Data Object elements must be contained within Process or Sub-Process elements. Data Object elements are visible in a Process diagram.

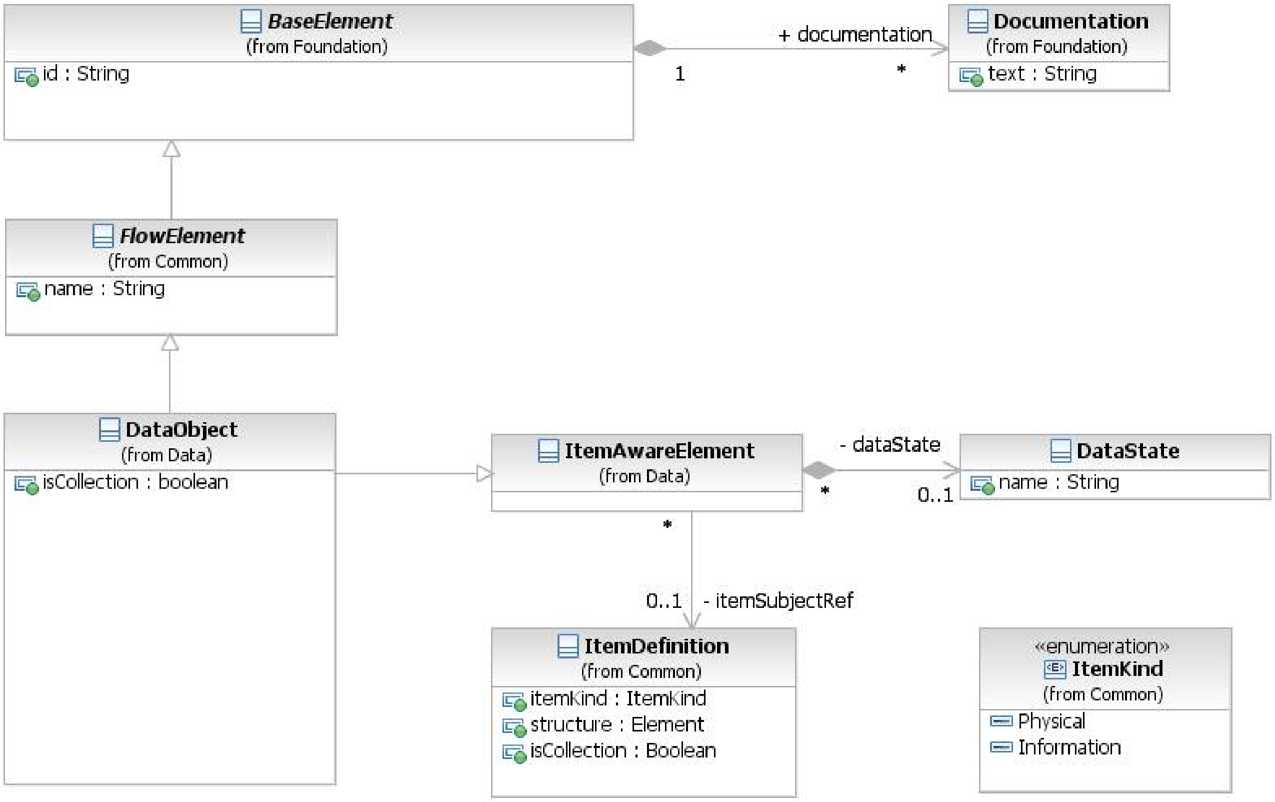


Figure 10-47 – DataObject class diagram

The DataObject element inherits the attributes and model associations of FlowElement (see Table 8-45) and ItemAwareElement (Table 10-46). Table 10-47 presents the additional attributes of the DataObject element:

Table 10-47 – DataObject attributes

Attribute Name	Description/Usage
isCollection: Boolean = False	Defines if the Data Object represents a collection of elements. This is a projection of the same attribute of the referenced ItemDefinition.

States

Data Object elements can optionally reference a DataState element, which is the state of the data contained in the Data Object (see an example of DataStates used for Data Objects in Figure 7-8). The definition of these states, e.g. possible values and any specific semantic are out of scope of this specification.

Therefore, BPMN adopters can use the State element and the BPMN extensibility capabilities to define their states.

The DataState element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-48 presents the additional attributes and model associations of the DataObject element:

Table 10-48 – DataState attributes and model associations

Attribute Name	Description/Usage
name: String	Defines the name of the DataState.

Data Objects representing a Collection of Data

A DataObject element that references an ItemDefinition marked as *collection* has to be visualized differently, compared to single *instance* data structures. The notation looks as follows:

Single *instance* (see Figure 10-48)

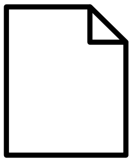


Figure 10-48 – A DataObject

Collection (see Figure 10-49)

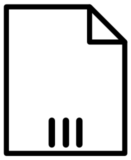


Figure 10-49 – A DataObject that is a collection

Visual representations of Data Objects

Data Object can appear multiple times in a Process diagram. Each of these appearances references the same Data Object *instance*. Multiple occurrences of a Data Object in a diagram are allowed to simplify diagram connections.

Lifecycle and Visibility

The lifecycle of a Data Object is tied to the lifecycle of its parent Process or Sub-Process. When a Process or Sub-Process is instantiated, all Data Objects contained within it are also instantiated. When a Process or Sub-Process *instance* is disposed, all Data Object *instances* contained within it are also disposed. At this point the data within these *instances* are no longer available.

The visibility of a Data Object is driven by its lifecycle. The data within a Data Object can only be accessed when there is guaranteed to be a live Data Object *instance* present. As a result, a Data Object can only be accessed by its immediate parent (Process or Sub-Process), or by its sibling Flow Elements and their children.

For example: Consider the follow structure.

- Process A
    - Data object 1
    - Task A
    - Sub-process A
        - Data object 2
        - Task B
    - Sub-process B
        - Data object 3
        - Sub-process C
            - Data object 4
            - Task C
        - Task D

“Data object 1” is visible to: “Process A,” “Task A,” “Sub-Process A,” “Task B,” “Sub-Process B,” “Sub-Process C,” “Task C,” and “Task D.”

“Data object 2” is visible to: “Sub-Process A” and “Task B.”

“Data object 3” is visible to: “Sub-Process B,” “Sub-Process C,” “Task C,” and “Task D.”

“Data object 4” is visible to: “Sub-Process C” and “Task C.”

Data Stores

A DataStore provides a mechanism for Activities to retrieve or update stored information that will persist beyond the scope of the Process. The same DataStore can be visualized, through a Data Store Reference, in one (1) or more places in the Process.

The Data Store Reference is an ItemAwareElement and can thus be used as the source or target for a Data Association. When data flows into or out of a Data Store Reference, it is effectively flowing into or out of the DataStore that is being referenced.

The notation looks as follows:

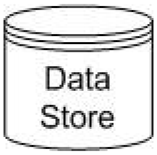


Figure 10-50 – A Data Store

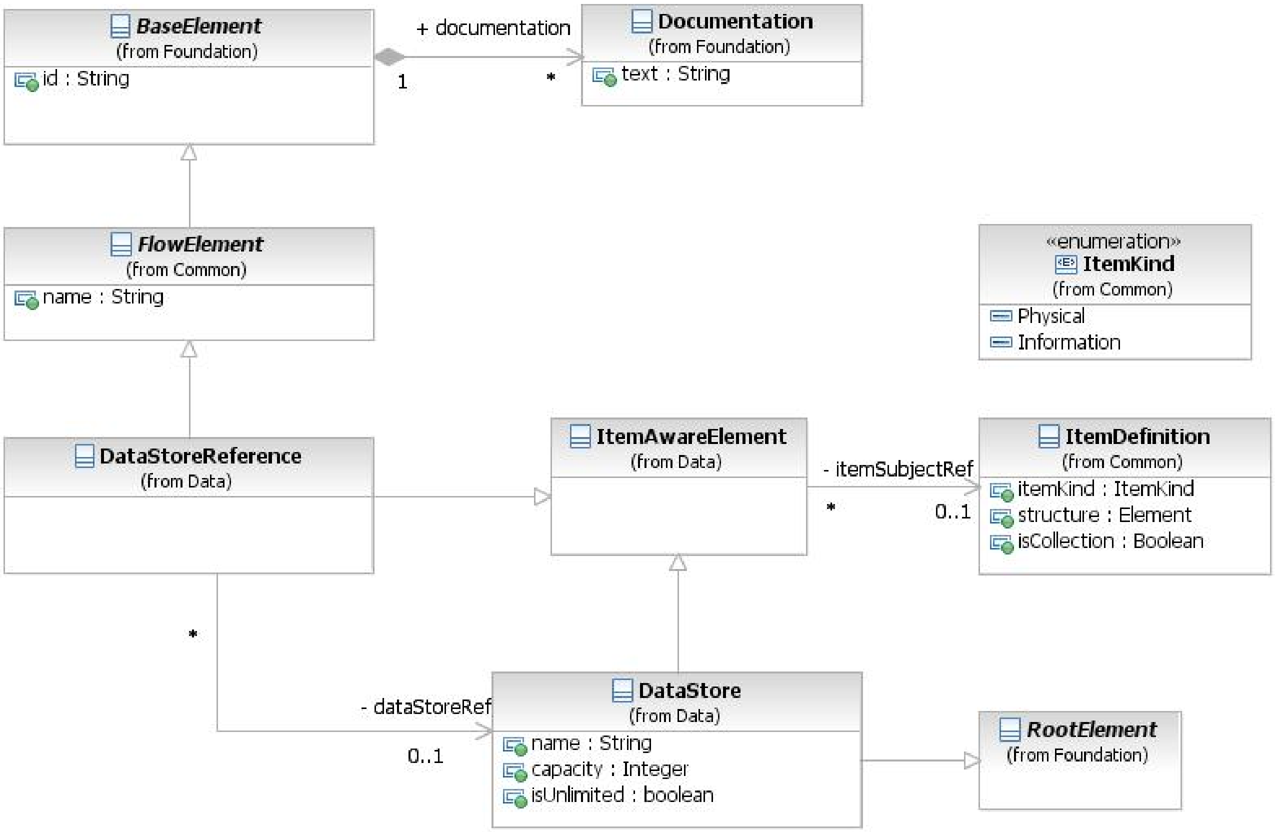


Figure 10-51 – DataStore class diagram

The `DataStore` element inherits the attributes and model associations of `FlowElement` (see Table 8-45) through its relationship to `RootElement`, and `ItemAwareElement` (see Table 10-46). Table 10-49 presents the additional attributes of the `DataStore` element:

Table 10-49 – Data Store attributes

Attribute Name	Description/Usage
name: string	A descriptive name for the element.
capacity: Integer [0..1]	Defines the `capacity` of the `Data Store`. This is not needed if the `isUnlimited` attribute is set to true
isUnlimited: Boolean = False	If `isUnlimited` is set to true, then the capacity of a `Data Store` is set as unlimited and will override any value of the `capacity` attribute.

The Data Store Reference element inherits the attributes and model associations of FlowElement (see Table 8-45) and ItemAwareElement (see Table 10-46). Table 10-49 presents the additional model associations of the Data Store Reference element:

Table 10-50 – Data Store attributes

Attribute Name	Description/Usage
dataStoreRef: DataStore	Provides the reference to a global DataStore.

Properties

Properties, like Data Objects, are item-aware elements. But, unlike Data Objects, they are not visible within a Process diagram. Certain flow elements may contain properties, in particular only Processes, Activities and Events may contain Properties

The Property class is a DataElement element that acts as a container for data associated with flow elements. Property elements must be contained within a FlowElement. Property elements are NOT visible in a Process diagram.

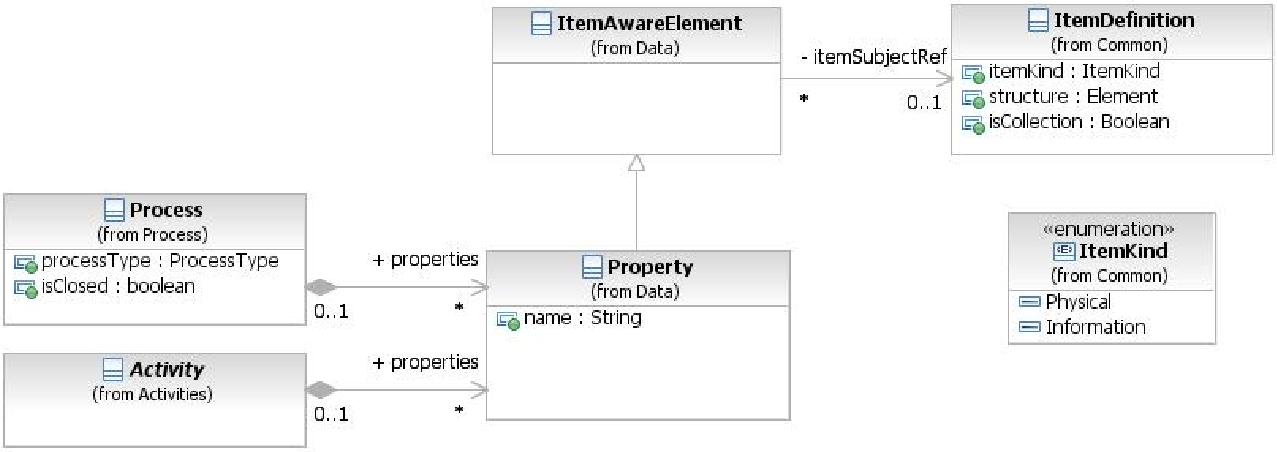


Figure 10-52 – Property class diagram

The Property element inherits the attributes and model associations of ItemAwareElement (Table 10-46). Table 10-47 presents the additional attributes of the Property element:

Table 10-51 – Property attributes

Attribute Name	Description/Usage
name: String	Defines the name of the Property.

Lifecycle and Visibility

The lifecycle of a Property is tied to the lifecycle of its parent Flow Element. When a Flow Element is instantiated, all Properties contained by it are also instantiated. When a Flow Element *instance* is disposed, all Property *instances* contained by it are also disposed. At this point the data within these instances are no longer available.

The visibility of a Property is driven by its lifecycle. The data within a Property can only be accessed when there is guaranteed to be a live Property *instance* present. As a result, a Property can only be accessed by its parent Flow Element or, when its parent Flow Element is a Process or Sub-Process, then by the immediate children of that Process or Sub-Process.

For example: Consider the follow structure.

- Process A
    - Task A
    - Sub-Process A
        - Task B
    - Sub-Process B
        - Sub-Process C
            - Task C
        - Task D

The Properties of “Process A” are visible to: All elements (including children elements) of this Process

The Properties of “Sub-Process A” are visible to: “Sub-Process A” and “Task B.”

The Properties of “Task C” are visible to: “Task C.”

Data Inputs and Outputs

Activities and Processes often required data in order to execute. In addition they may produce data during or as a result of execution. Data requirements are captured as Data Inputs and Input Sets. Data that is produced is captured using Data Outputs and Output Sets. These elements are aggregated in a InputOutputSpecification class.

Certain Activities and CallableElements contain a InputOutputSpecification element to describe their data requirements. Execution semantics are defined for the input/output specification and they apply the same way to all elements that extend it. Not every Activity type defines inputs and outputs, only Tasks, CallableElements (Global Tasks and Processes) can define their data requirements.

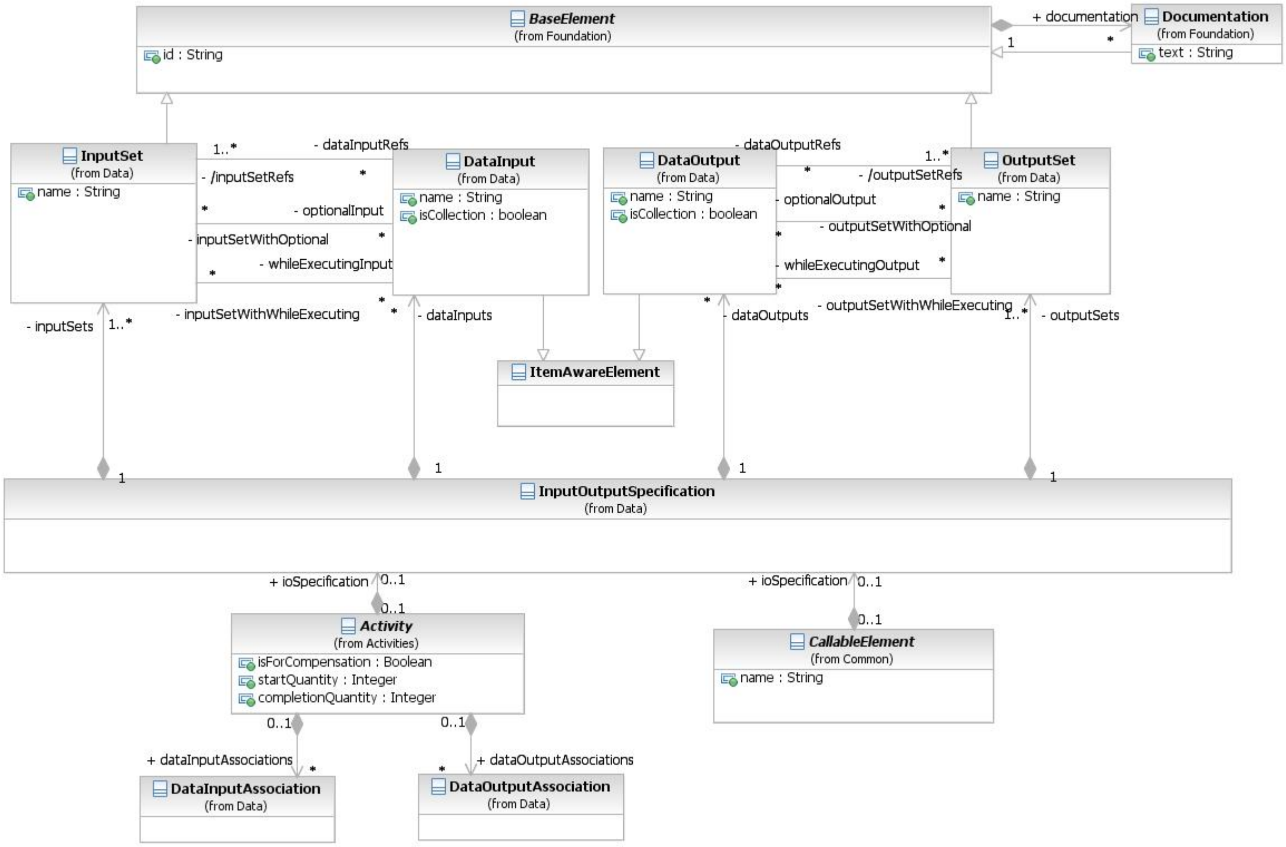


Figure 10-53 – InputOutputSpecification class diagram

The InputOutputSpecification element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-52 presents the additional attributes and model associations of the InputOutputSpecification element:

Table 10-52 – InputOutputSpecification Attributes and Model Associations

Attribute Name	Description/Usage
inputSets: InputSet [1..*]	A reference to the InputSets defined by the InputOutputSpecification. Every InputOutputSpecification must define at least one InputSet.
outputSets: OutputSet [1..*]	A reference to the OutputSets defined by the InputOutputSpecification. Every Data Interface must define at least one OutputSet.
dataInputs: DataInput [0..*]	An optional reference to the Data Inputs of the InputOutputSpecification. If the InputOutputSpecification defines no Data Input, it means no data is required to start the Activity.
dataOutputs: DataOutput [0..*]	An optional reference to the Data Outputs of the InputOutputSpecification. If the InputOutputSpecification defines no Data Output, it means no data is required to finish the Activity.

DataInput

A DataInput is a declaration that a particular kind of data will be used as input of the InputOutputSpecification. There may be multiple data inputs associated with an InputOutputSpecification.

The DataInput is an item-aware element. DataInput elements may appear in a Process diagram to show the inputs to the Process as whole, which are passed along as the inputs of Activities by DataAssociations.

- DataInputs have the same notation as DataObjects, except MUST contain a small, unfilled block arrow (see Figure 10-54).
- DataInputs MUST NOT have *incoming* DataAssociations.

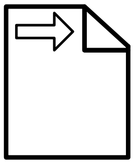


Figure 10-54 – A DataInput

The "**optional**" attribute defines if a DataInput is valid even if the state is "**unavailable**". The default value is *false*. If the value of this attribute is *true*, then the execution of the Activity will not begin until a value is assigned to the DataInput element, through the corresponding Data Associations.

States

DataInput elements can optionally reference a DataState element, which is the state of the data contained in the DataInput. The definition of these states, e.g. possible values, and any specific semantics are out of scope of this specification. Therefore, BPMN adopters can use the DataState element and the BPMN extensibility capabilities to define their states.

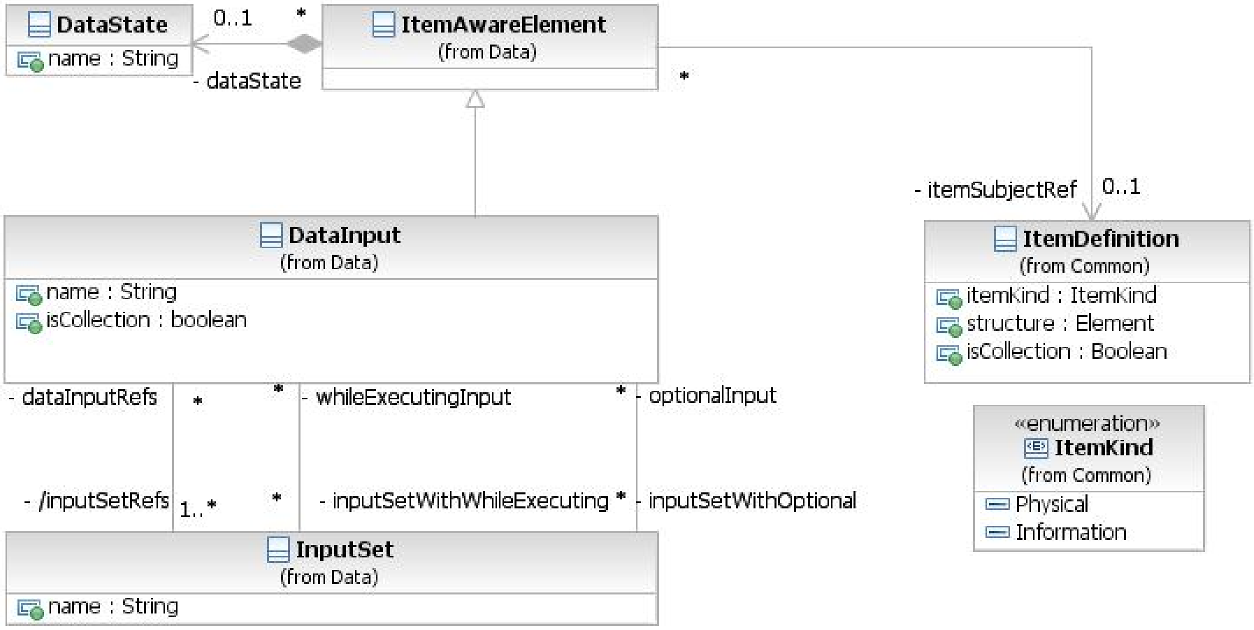


Figure 10-55 – Data Input class diagram

The DataInput element inherits the attributes and model associations of BaseElement (see Table 8-5) and ItemAwareElement (Table 10-46). Table 10-53 presents the additional attributes and model associations of the DataInput element:

Table 10-53 – DataInput attributes and model associations

Attribute Name	Description/Usage
name: string	A descriptive name for the element.
inputSetRefs: InputSet [1..*]	A DataInput is used in one (1) or more InputSets. This attribute is derived from the InputSets.
inputSetwithOptional: InputSet [0..*]	Each InputSet that uses this DataInput can determine if the Activity can start executing with this DataInput state in "**unavailable**". This attribute lists those InputSets.
inputSetWithWhileExecuting: Inputset [0..*]	Each InputSet that uses this DataInput can determine if the Activity can evaluate this DataInput while executing. This attribute lists those InputSets.
isCollection: Boolean = False	Defines if the DataInput represents a collection of elements. This is a projection of the same attribute of the referenced ItemDefinition.

DataOutput

A DataOutput is a declaration that a particular kind of data may be produced as output of the InputOutputSpecification. There may be multiple data outputs associated with a InputOutputSpecification.

The DataOutput is an item-aware element. DataOutput elements appear in a Process diagram to show the outputs of the Process as whole, which are passed along from the outputs of Activities by DataAssociations.

- DataOutputs have the same notation as DataObjects, except MUST contain a small, filled block arrow (see Figure 10-54).
- DataOutputs MUST NOT have *outgoing* DataAssociations.

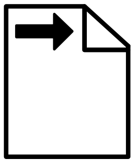


Figure 10-56 – A Data Output

States

DataOutput elements can optionally reference an DataState element, which is the state of the data contained in the DataOutput. The definition of these states, e.g. possible values, and any specific semantics are out of scope of this specification. Therefore, BPMN adopters can use the DataState element and the BPMN extensibility capabilities to define their states.

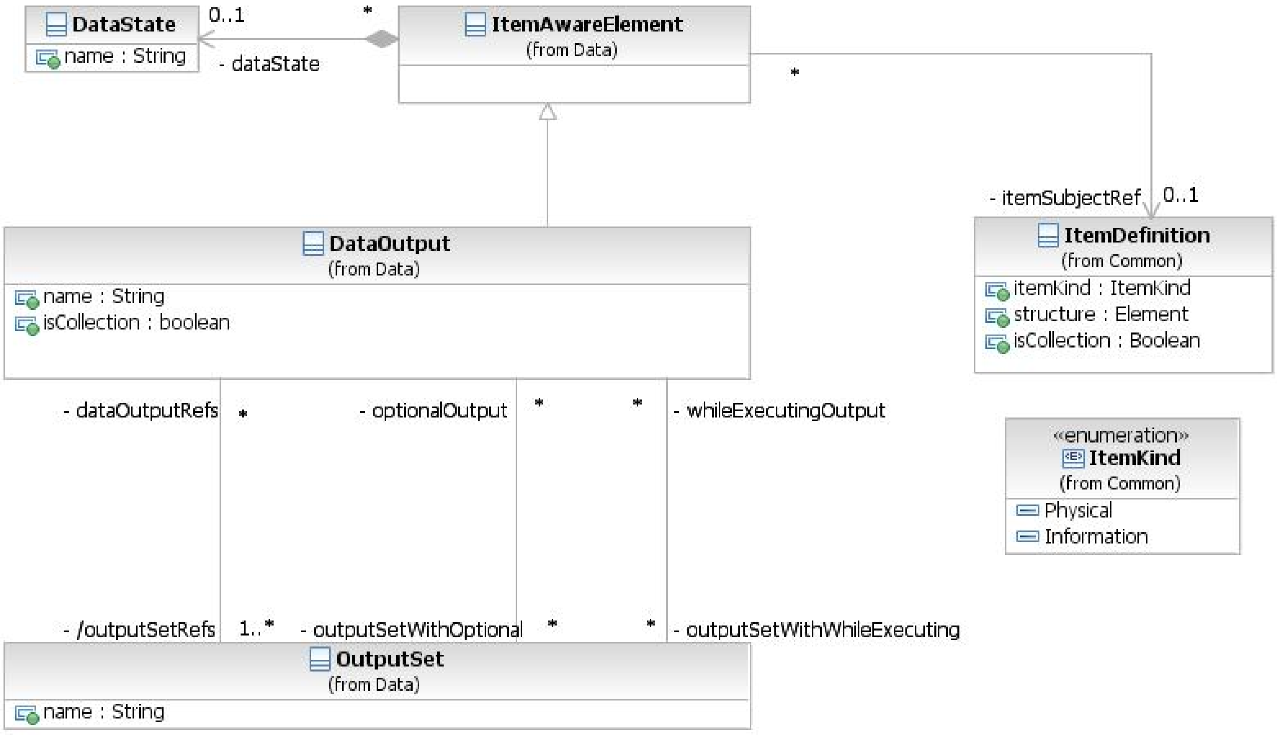


Figure 10-57 – Data Output class diagram

The DataOutput element inherits the attributes and model associations of BaseElement (see Table 8-5) and ItemAwareElement (Table 10-46). Table 10-54 presents the additional attributes and model associations of the DataInput element:

Table 10-54 – DataOutput attributes and associations

Attribute Name	Description/Usage
name: String	A descriptive name for the element.
outputSetRefs: OutputSet [1..*]	A DataOutput is used in one (1) or more OutputSets. This attribute is derived from the OutputSets.
outputSetwithOptional: OutputSet [0..*]	Each OutputSet that uses this DataOutput can determine if the Activity can complete executing without producing this DataInput. This attribute lists those OutputSets.
outputSetWithWhileExecuting: OutputSet [0..*]	Each OutputSet that uses this DataInput can determine if the Activity can produce this DataOutput while executing. This attribute lists those OutputSets.
isCollection: Boolean = False	Defines if the DataOutput represents a collection of elements. This is a projection of the same attribute of the referenced ItemDefinition.

The following describes the mapping of data inputs and outputs to the specific Activity implementations:

Service Task Mapping

There is a single data input that has a ItemDefinition equivalent to the one defined by the Message referred by the inMessageRef attribute of the operation.

In the case the operation defines output Messages, there is a single data output that has an ItemDefinition equivalent to the one defined by Message referred by the outMessageRef attribute of the operation.

User Task Mapping

User Tasks have access to the Data Input, Data Output and the data aware elements available in the scope of the User Task.

Call Activity Mapping

Since Call Activities rely in the callable element being invoked, the data inputs and outputs of the Call Activity must match with the data inputs, inputsets and outputs, outputsets defined in the callable element. The data inputs and outputs of the Call Activity are mapped to the corresponding data inputs and output of the Callable Element without any explicit data association.

Script Task Mapping

Script Tasks have access to the Data Input , Data Output and the data aware elements available in the scope of the Script Task.

InputSet

An InputSet is a collection of DataInput elements that together define a valid set of data inputs for a InputOutputSpecification. A InputOutputSpecification must have at least one InputSet element. An InputSet may reference zero or more DataInput elements. A single DataInput may be associated with multiple InputSet elements, but it must always be referenced by at least one InputSet.

An "empty" InputSet, one that references no DataInput elements, signifies that the Activity requires no data to start executing (this implies that either there are no data inputs or they are referenced by another input set).

InputSet elements are contained by InputOutputSpecification elements; the order in which these elements are included defines the order in which they will be evaluated.

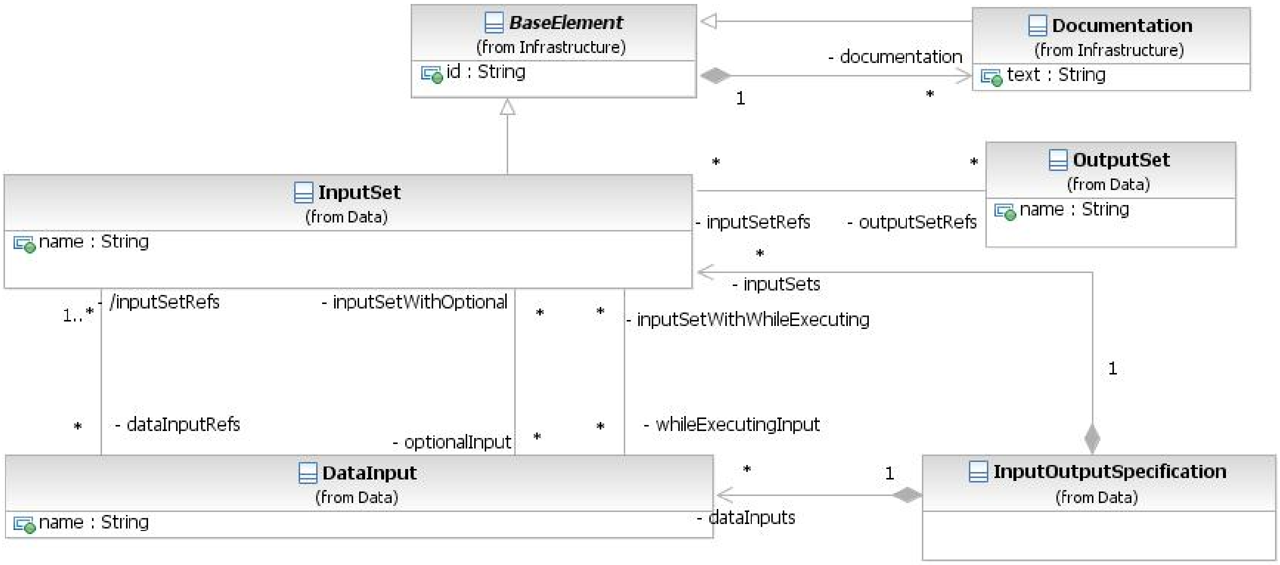


Figure 10-58 – InputSet class diagram

The InputSet element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-55 presents the additional attributes and model associations of the InputSet element:

Table 10-55 – InputSet attributes and model associations

Attribute Name	**Description/Usage**
name: string [0..1]	A descriptive name for the input set.
dataInputRefs: DataInput [0..*]	The DataInput elements that collectively make up this data requirement.
optionalInput: DataInput [0..*]	The DataInput elements that are a part of the InputSet that can be in the state of "unavailable" when the Activity starts executing. This association MUST NOT reference a DataInput that is not listed in the dataInputRefs.
whileExecutingInput: DataInput [0..*]	The DataInput elements that are a part of the InputSet that can be evaluated while the Activity is executing. This association MUST NOT reference a DataInput that is not listed in the dataInputRefs.
outputSetRefs: OutputSet [0..*]	Specifies an Input/Output rule that defines which OutputSet is expected to be created by the Activity when this InputSet became valid. This attribute is paired with the inputSetRefs attribute of OutputSets. This combination replaces the IORules attribute for Activities in BPMN 1.2.

OutputSet

An OutputSet is a collection of DataOutputs elements that together may be produced as output from an Activity or Event. An InputOutputSpecification element must define at least OutputSet element. An OutputSet may reference zero or more DataOutput elements. A single DataOutput may be associated with multiple OutputSet elements, but it must always be referenced by at least one OutputSet.

An "empty" OutputSet, one that is associated with no DataOutput elements, signifies that the ACTIVITY may produce no data.

The implementation of the element where the OutputSet is defined must determine the OutputSet that will be produced. So it is up to the Activity implementation or the Event, to define which OutputSet will be produced.

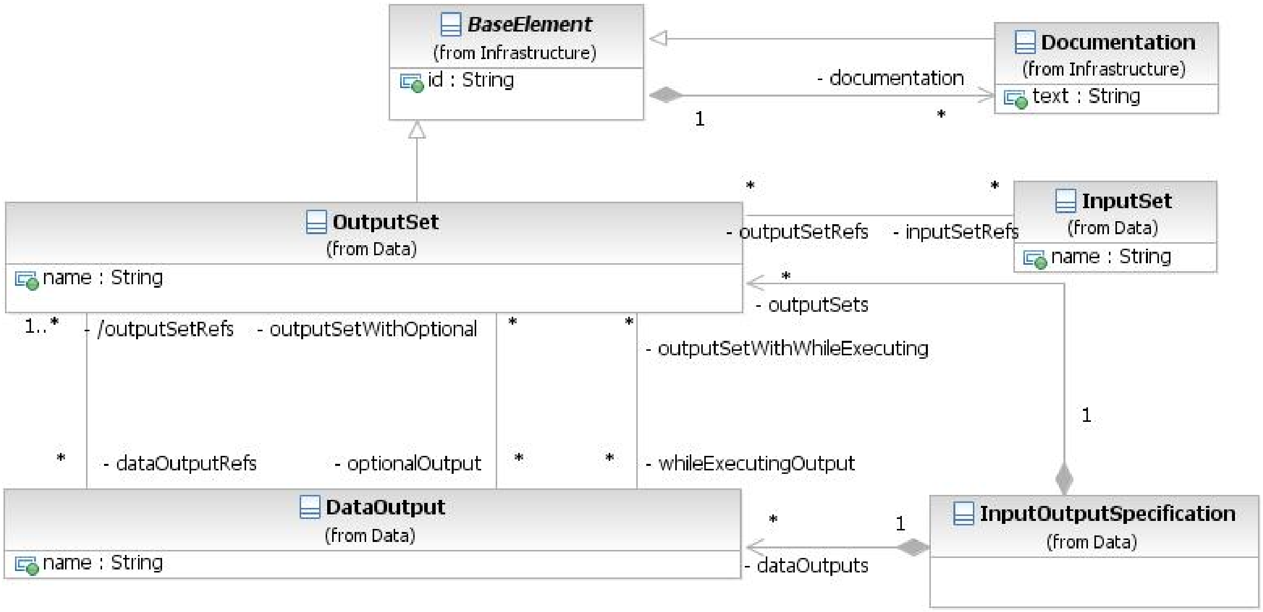


Figure 10-59 – OutputSet class diagram

The `OutputSet` element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 10-56 presents the additional attributes and model associations of the `OutputSet` element:

Table 10-56 – OutputSet attributes and model associations

Attribute Name	Description/Usage
name: string [0..1]	A descriptive name for the input set.
dataOutputRefs: DataOutput [0..*]	The DataOutput elements that may collectively be outputted.
optionalOutput: DataInput [0..*]	The `DataOutput` elements that are a part of the `OutputSet` that do not have to be produced when the `Activity` completes executing. This association MUST NOT reference a `DataOutput` that is not listed in the `dataOutputRefs`.
whileExecutingOutput: DataInput [0..*]	The `DataOutput` elements that are a part of the `OutputSet` that can be produced while the `Activity` is executing. This association MUST NOT reference a `DataOutput` that is not listed in the `dataOutputRefs`.
inputSetRefs: InputSet [0..*]	Specifies an Input/Output rule that defines which `InputSet` has to become valid to expect the creation of this `OutputSet`. This attribute is paired with the `outputSetRefs` attribute of `InputSets`. This combination replaces the `IORules` attribute for `Activities` in BPMN 1.2.

Data Associations

Data Associations are used to move data between Data Objects, Properties, and *inputs* and *outputs* of Activities, Processes, and GlobalTasks. *Tokens* do not flow along a Data Association, and as a result they have no direct effect on the flow of the Process.

The purpose of retrieving data from Data Objects or Process Data Inputs is to fill the Activities *inputs* and later push the *output* values from the execution of the Activity back into Data Objects or Process Data Outputs.

DataAssociation

The DataAssociation class is a BaseElement contained by an Activity or Event, used to model how data is pushed into or pulled from item-aware elements. DataAssociation elements have one or more sources and a target; the source of the association is copied into the target.

The ItemDefinition from the souceRef and targetRef must have the same ItemDefinition or the DataAssociation MUST have a transformation Expression that transforms the source ItemDefinition into the target ItemDefinition.

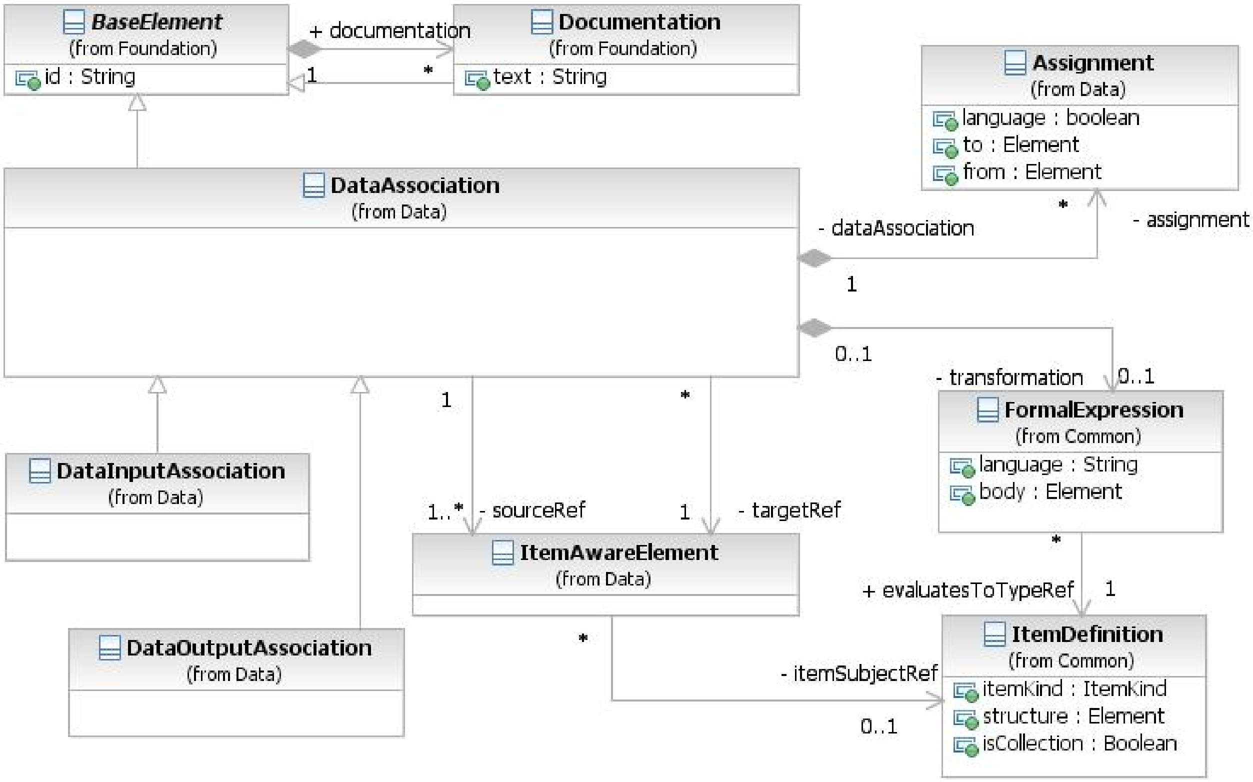


Figure 10-60 – DataAssociation class diagram

Optionally, Data Associations can be visually represented in the diagram by using the Association connector style.

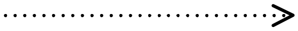


Figure 10-61 – A Data Association

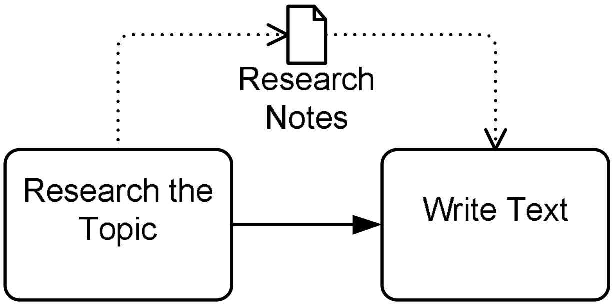


Figure 10-62 – A Data Association used for an Outputs and Inputs into an Activities

The core concepts of a DataAssociation are that they have sources, a target and an optional transformation.

When a data association is “executed”, data is copied to the target. What is copied depends if there is a transformation defined or not.

If there is no transformation defined or referenced, then only one source must be defined, and the contents of this source will be copied into the target.

If there is a transformation defined or referenced, then this transformation expression will be evaluated and the result of the evaluation is copied into the target. There can be zero to many sources defined in this case, but there is no requirement that these sources are used inside the expression.

In any case, sources are used to define if the data association can be “executed”, if any of the sources is in the state of “unavailable”, then the data association cannot be executed, and the Activity or Event where the data association is defined must wait until this condition is met.

Data Associations are always contained within another element that defines when these data associations are going to be executed. Activities define two (2) sets of data associations, while Events define only 1 (one).

For Events, there is only one set, but they are used differently for *catch* or *throw* Events. For a *catch* Event, data associations are used to push data from the Message received into Data Objects and properties. For a *throw* Event, data associations are used to fill the Message that is being thrown.

As DataAssociation are used in different stages of the Process and Activity lifecycle, the possible sources and targets vary according to that stage. This defines the scope of possible elements that can be referenced as source and target. For example: when an Activity starts executing, the scope of valid targets include the Activity data inputs, while at the end of the Activity execution, the scope of valid sources include Activity data outputs.

The DataAssociation element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-57 presents the additional model associations of the DataAssociation element:

Table 10-57 – DataAssociation model associations

Attribute Name	Description/Usage
transformation: Expression [0..1]	Specifies an optional transformation expression. The actual scope of visible data for that expression is defined by the source and target of the specific data association types.
assignment: Assignment [0..*]	Specifies one or more data elements Assignments. By using an Assignment, single data structure elements can be assigned from the source structure to the target structure.
sourceRef: ItemAwareElement [1..*]	Identifies the source of the data association. The source must be an ItemAwareElement.
targetRef: ItemAwareElement	Identifies the target of the data association. The target must be an ItemAwareElement

Assignment

The Assignment class is used to specify a simple mapping of data elements using a specified expression language.

The default expression language for all expressions is specified in the Definitions element, using the expressionLanguage attribute. It can also be overridden on each individual Assignment using the same attribute.

The Assignment element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-58 presents the additional attributes of the Assignment element:

Table 10-58 – Assignment attributes

Attribute Name	Description/Usage
language: string [0..1]	When included, this will override the Expression language specified in the Definitions.
from: Element	The body of the Expression that evaluates the source of the Assignment.
to: Element	The body of the Expression that defines the actual Assignment operation and the target data element.

DataInputAssociation

The DataInputAssociation can be used to associate a item-aware element with a DataInput contained in an Activity. The source of such a DataAssociation can be every item-aware element visible to the current scope, e.g. a Data Object, a Property or an Expression.

The DataInputAssociation element inherits the attributes and model associations of FlowElement (see Table 10-57), but does not contain any additional attributes or model associations.

DataOutputAssociation

The DataOutputAssociation can be used to associate a DataOutput contained within an ACTIVITY with any item-aware element visible to the scope the association will be executed in. The target of such a DataAssociation can be every item-aware element visible to the current scope, e.g. a Data Object, a Property or an Expression.

The DataOutputAssociation element inherits the attributes and model associations of FlowElement (see Table 10-57), but does not contain any additional attributes or model associations.

Data Objects associated with a Sequence Flow

Figure 10-63 repeats Figure 10-62, above, and shows how Data Associations are used to represent inputs and outputs of Activities.

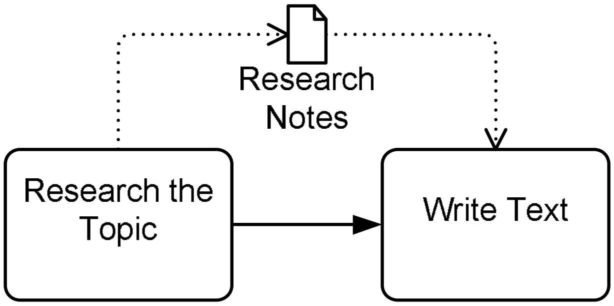


Figure 10-63 – A Data Object shown as an output and an inputs

Alternatively, Data Objects may be directly associated with a Sequence Flow connector (see Figure 10-64) to represent the same input/output relationships. This is a visual short cut that normalizes two Data Associations (e.g., as seen in Figure 10-63, above): one from a item-aware element (e.g., an Activity) contained by the source of the Sequence Flow, connecting to the Data Object; and the other from the Data Object connecting to a item-aware element contained by the target of the Sequence Flow.

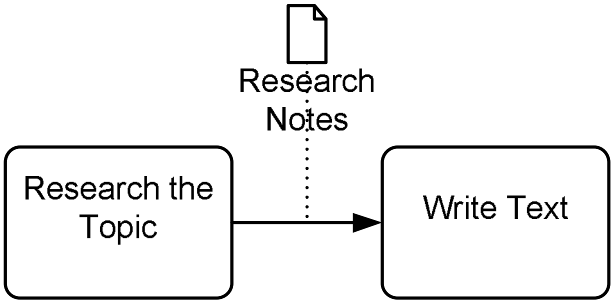


Figure 10-64 – A Data Object associated with a Sequence Flow

10.3.2. Execution Semantics for Data

When an element that defines a InputOutputSpecification is ready to begin execution by means of Sequence Flow or Event being caught, the inputs of the interface are filled with data coming from elements in the context, such as Data Objects or Properties. The way to represent these assignments is the Data Association elements.

Each defined InputSet element will be evaluated in the order they are included in the InputOutputSpecification.

For each InputSet, the data inputs it references will be evaluated if it is valid.

All data associations that define as target the data input will be evaluated, and if any of the sources of the data association is "**unavailable**", then the InputSet is "**unavailable**" and the next InputSet is evaluated.

The first InputSet where all data inputs are "available" (by means of data associations) is used to start the execution of the Activity. If no InputSet is "available", then the execution will wait until this condition is met.

The time and frequency of when and how often this condition is evaluated is out of scope this specification. Implementations will wait for the sources of data associations to become available and then re-evaluate the InputSets.

10.3.3. Usage of Data in XPath Expressions

BPMN extensibility mechanism enables the usage of various languages for expressions and queries. This section describes how XPath is used in BPMN. It introduces a mechanism to access BPMN Data Objects, BPMN *Properties*, and various *instance* attributes from XPath expressions.

The visibility to the Expression language is defined based on the entities visibility to the Activity that contains the expression. All elements visible from the enclosing element of an XPath expression must be made available to the XPath processor.

BPMN Data Objects and BPMN *Properties* are defined using ItemDefinition. The XPath binding assumes that the ItemDefinition is either an XSD complex type or an XSD element. If XSD element is used it must be manifested as a node-set XPath variable with a single member node. If XSD complex type is used it must be manifested as a node-set XPath variable with one member node containing the anonymous document element that contains the actual value of the BPMN Data Object or *Property*.

Access to BPMN Data Objects

The table below introduces an XPath function used to access BPMN Data Objects. Argument processName names Process and is of type string. Argument dataObjectName names Data Object and is of type string. It must be a literal string.

Table 10-59 – XPath Extension Function for Data Objects

XPath Extension Function	Description/Usage
Element getDataObject (‘processName’, ‘dataObjectName’)	This extension function returns value of submitted Data Object. Argument processName is optional. If omitted, the process enclosing the activity that contains the expression is assumed. In order to access Data Objects defined in a parent process the processName must be used. Otherwise it must be omitted.

Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.

Access to BPMN Data Input and Data Output

The table below introduces XPath functions used to access BPMN Data Inputs and BPMN Data Outputs. Argument dataInputName names a Data Input and is of type string. Argument dataOutput names a Data Output and is of type string.

Table 10-60 – XPath Extension Function for Data Inputs and Data Outputs

XPath Extension Function	Description/Usage
Element getDataInput (‘dataInputName’)	This extension function returns the value of the submitted Data Input.
Element getDataOutput('dataOutputName')	This extension function returns the value of the submitted Data Output.

Access to BPMN Properties

The table below introduces XPath functions used to access BPMN *Properties*.

Argument processName names Process and is of type string. Argument propertyName names property and is of type string. Argument activityName names Activity and is of type string. Argument eventName names Event and is of type string. These strings must be literal strings. The XPath extension functions return value of the submitted property.

Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.

Table 10-61 – XPath Extension Functions for Properties

XPath Extension Function	Description/Usage
Element getProcessProperty (‘processName’, ‘propertyName’)	This extension function returns value of submitted Process property. Argument processName is optional. If omitted, the process enclosing the activity that contains the expression is assumed. In order to access Properties defined in a parent process the processName must be used. Otherwise it must be omitted.
Element getActivityProperty (‘activityName’, ‘propertyName’)	This extension function returns value of submitted Activity property.
Element getEventProperty ‘eventName’, ‘propertyName’)	This extension function returns value of submitted Event property.

For BPMN Instance Attributes

The table below introduces XPath functions used to access `BPMN Instance` *Attributes*.

Argument `processName` names `Process` and is of type `string`. Argument `attributeName` names `Instance` attribute and is of type `string`. Argument `activityName` names `Activity` and is of type `string`. These `strings` must be literal `strings`

These functions return value of the submitted *instance* `Activity`. Because XPath 1.0 functions do not support returning faults, an empty node set is returned in the event of an error.

Table 10-62 – XPath Extension Functions for Instance Attributes

XPath Extension Function	Description/Usage
Element getProcessInstanceAttribute (‘processName’,‘attributeName’)	This extension function returns value of submitted Process instance attribute. Argument processName is optional. If omitted, the process enclosing the activity that contains the expression is assumed. In order to access Instance Attributes of a parent process the processName must be used. Otherwise it must be omitted.
Element getChoreographyInstanceAttribute (‘attributeName’)	This extension function returns value of submitted Choreography instance attribute.
Element getActivityInstanceAttribute (‘activityName’, ‘attributeName’)	This extension function returns value of submitted Activity instance attribute. User Task and Loop are examples of activities.

10.3.4. XML Schema for Data

Table 10-63 – Assignment XML schema

```
<xsd:element name="assignment" type="tAssignment" />
<xsd:complexType name="tAssignment">
        <xsd:complexContent>
                <xsd:extension base="tBaseElement">
                        <xsd:sequence>
                                <xsd:element name="from"
                                type="tBaseElementWithMixedContent" minOccurs="1"
                                maxOccurs="1"/>
                                <xsd:element name="to" type="tBaseElementWithMixedContent"
                                minOccurs="1" maxOccurs="1"/>
                        </xsd:sequence>
                        <xsd:attribute name="language" type="xsd:anyURI"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-64 – DataAssociation XML schema

```
<xsd:element name="dataAssociation" type="tDataAssociation" />
<xsd:complexType name="tDataAssociation" abstract="true">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="transformation" type="tFormalExpression" minOccurs="0"
                        maxOccurs="1"/>
                <xsd:element ref="assignment" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-65 – DataInput XML schema

```
<xsd:element name="dataInput" type="tDataInput" />
<xsd:complexType name="tDataInput">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string" />
            <xsd:attribute name="itemSubjectRef" type="xsd:QName" />
            <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
            <xsd:attribute name="dataState" type="xsd:IDREF"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-66 – DataInputAssociation XML schema

```
<xsd:element name="dataInputAssociation" type="tDataInputAssociation" />
<xsd:complexType name="tDataInputAssociation">
      <xsd:complexContent>
        <xsd:extension base="tDataAssociation">
            <xsd:sequence>
                <xsd:element name="sourceRef" type="xsd:IDREF" minOccurs="1"
                        maxOccurs="unbounded"/>
                <xsd:element name="targetRef" type="xsd:IDREF" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-67 – InputOutputSpecification XML schema

```
<xsd:element name="ioSpecification" type="tInputOutputSpecification" />
<xsd:complexType name="tInputOutputSpecification">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="inputSet" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="outputSet" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-68 – DataObject XML schema

```
<xsd:element name="dataObject" type="tDataObject" />
<xsd:complexType name="tDataObject">
    <xsd:complexContent>
        <xsd:extension base="tFlowElement">
            <xsd:sequence>
                <xsd:element ref="dataState" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
            <xsd:attribute name="isCollection" type="xsd:boolean"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-69 – DataState XML schema

```
<xsd:element name="dataState" type="tDataState" />
<xsd:complexType name="tDataState">
        <xsd:complexContent>
          <xsd:extension base="tBaseElement">
               <xsd:attribute name="name" type="xsd:string"/>
          </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-70 – DataOutput XML schema

```
<xsd:element name="dataOutput" type="tDataOutput" />
<xsd:complexType name="tDataOutput">
        <xsd:complexContent>
          <xsd:extension base="tBaseElement">
               <xsd:attribute name="name" type="xsd:string" />
               <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
               <xsd:attribute name="isCollection" type="xsd:boolean" default="false"/>
               <xsd:attribute name="dataState" type="xsd:IDREF"/>
          </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-71 – DataOutputAssociation XML schema

```
<xsd:element name="dataOutputAssociation" type="tDataOutputAssociation" />
<xsd:complexType name="tDataOutputAssociation">
        <xsd:complexContent>
          <xsd:extension base="tDataAssociation">
               <xsd:sequence>
<xsd:element name="sourceRef" type="xsd:IDREF" minOccurs="1" maxOccurs="unbounded"/>
<xsd:element name="targetRef" type="xsd:IDREF" minOccurs="1" maxOccurs="1"/>
               </xsd:sequence>
          </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-72 – InputSet XML schema

```
<xsd:element name="inputSet" type="tInputSet" />
<xsd:complexType name="tInputSet">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="dataInputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="optionalInputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="whileExecutingInputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="outputSetRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" />
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-73 – OutputSet XML schema

```
<xsd:element name="outputSet" type="tOutputSet" />
<xsd:complexType name="tOutputSet">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="dataOutputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="optionalOutputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="whileExecutingOutputRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element name="inputSetRefs" type="xsd:IDREF" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-74 – Property XML schema

```
<xsd:element name="property" type="tProperty" />
<xsd:complexType name="tProperty">
      <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="itemSubjectRef" type="xsd:QName"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

10.4. Events

An Event is something that “happens” during the course of a Process. These Events affect the flow of the Process and usually have a cause or an impact and in general require or allow for a reaction. The term “event” is general enough to cover many things in a Process. The start of an Activity, the end of an Activity, the change of state of a document, a Message that arrives, etc., all could be considered Events.

Events allow for the description of “event-driven” Processes. In these Processes, There are three main types of Events:

- Start Events (see page 244), which indicate where a Process will start.
- End Events (see page 252), which indicate where a path of a Process will end.
- Intermediate Events (see page 256), which indicate where something happens somewhere between the start and end of a Process.

Within these three types, Events come in two flavors:

- Events that *catch* a *Trigger*. All Start Events and some Intermediate Events are *catching* Events.
- Events that *throw* a *Result*. All End Events and some Intermediate Events are *throwing* Events that may eventually be *caught* by another Event. Typically the *Trigger* carries information out of the scope where the *throw* Event occurred into the scope of the catching Events. The *throwing* of a *trigger* may be either implicit as defined by this standard or an extension to it or explicit by a *throw* Event.

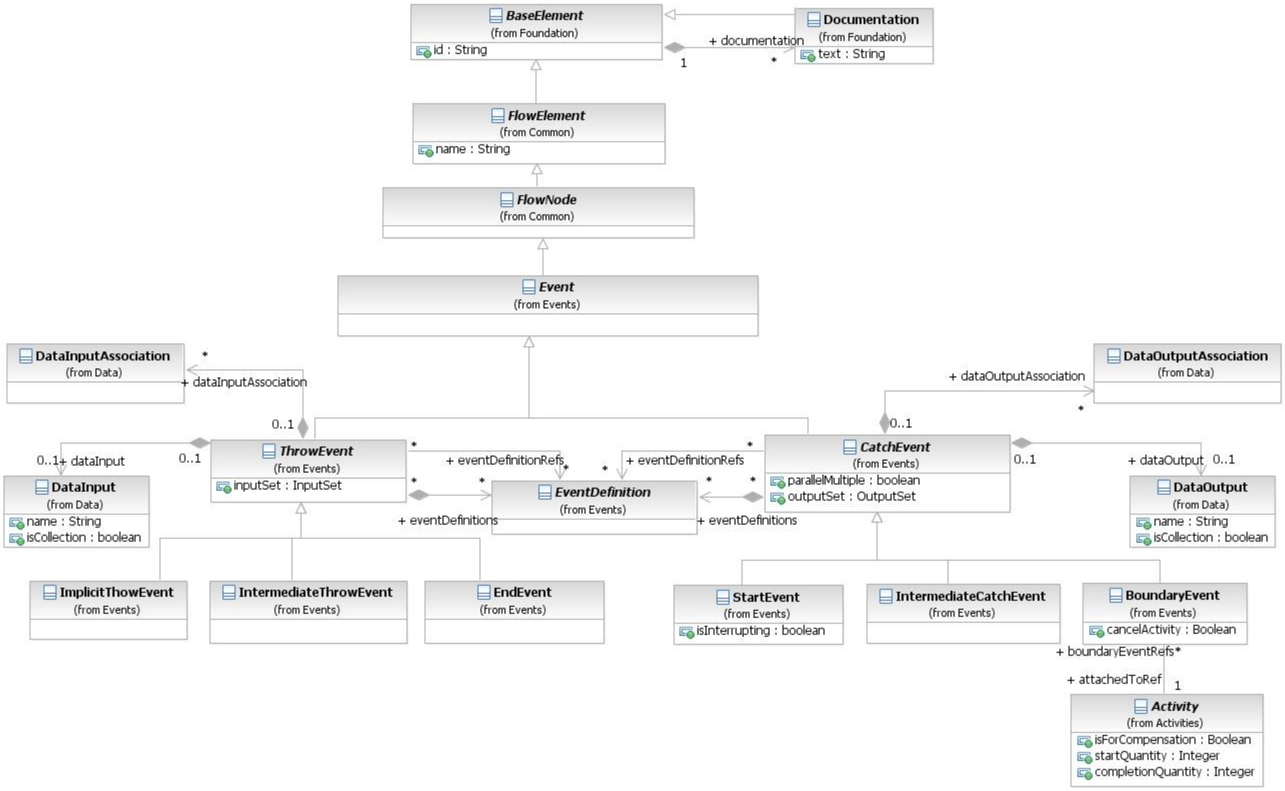


Figure 10-65 – The Event Class Diagram

10.4.1. Concepts

Depending on the type of the Event there are different strategies to forward the *trigger* to catching Events: publication, direct resolution, propagation, *cancellations*, and *compensations*.

With publication a *trigger* may be received by any catching Events in any scope of the system where the *trigger* is published. Events for which publication is used are grouped to Conversations. Published Events may participate in several Conversations. Messages are *triggers*, which are generated outside of the Pool they are published in. They typically describe B2B communication. When Messages need to reach a specific Process *instance*, *correlation* is used to identify the particular *instance*. Signals are *triggers* generated in the Pool they are published.

Timer and Conditional *triggers* are implicitly thrown. When they are activated they wait for a time based or status based condition respectively to *trigger* the *catch* Event.

A *trigger* that is propagated is forwarded from the location where the Event has been thrown to the innermost enclosing scope *instance* (e.g., Process level) which has an attached Event being able to catch the trigger. Error *triggers* are critical and suspend execution at the location of throwing. Escalations are non critical and execution continues at the location of throwing. If no catching Event is found for an error or escalation *trigger*, this *trigger* is unresolved.

Termination**,** *compensation*, and *cancellation* are directed towards a Process or a specific Activity *instance*. Termination indicates that all Activities in the Process or Activity should be immediately ended. This includes all *instances* of *multi-instances*. It is ended without *compensation* or *Event handling*.

Compensation of a successfully completed Activity triggers its *compensation handler*. The *compensation handler* is either user defined or implicit. The implicit *compensation handler* of a Sub Process calls all *compensation handlers* of its enclosed Activities in the reverse order of Sequence Flow dependencies. If *compensation* is invoked for an Activity that has not yet completed, or has not completed successfully, nothing happens (in particular, no error is raised).

Cancellation will terminate all running Activities and *compensate* all successfully completed Activities in the Sub-Process it is applied to. If the Sub-Process is a *Transaction*, the *Transaction* is rolled back.

Data Modeling and Events

Some Events (like the Message, Signal, Error, Escalation and Multiple Event) have the capability to carry data. Data Association is used to push data from a *Catch* Event to a Data Element. For *Throw* Events, a Data Association is used to fill the Event data from a Data Element (see page 211 for the definition of Data Elements).

Catch Event Data Association

The Data Association for a *Catch* Event is performed after the *trigger* of the *Catch* Event occurs (Message, Signal, Error, Escalation, Multiple data is available in the *Catch* Event). The Data Association assigns the data of the Event to a Data Element that is in the scope of the *Catch* Event. After that, Sequence Flow continues as usual.

For example, consider a *Receive* Message Intermediate Event; as soon as the Message is received by the Event, the Data Association is performed and the Message data is assigned for example to a Data Object of the Process.

Throw Event Data Association

The Data Association for a *Throw* Event is performed when the Sequence Flow arrives at the *Throw* Event. The Data Association assigns the data from a Data Element that is in the scope of the *Throw* Event to the Event data (Message, Signal, Error, Escalation, and Multiple). After that the *trigger* of the Event will occur.

For example, consider a Message End Event; when the Sequence Flow reaches the Message End Event, the data association of the Event is performed. The data association assigns data from a Data Element that is in the scope of the Message End Event to the Message. Then the Message is send to a *Participant*.

Common Catch Event attributes

The following table shows the common attributes for Catch Events.

The CatchEvent element inherits the attributes and model associations of FlowElement (see Table 8-45) through its relationship to the Event element (see page 103). Table 10-75 presents the additional attributes and model associations of the CatchEvent element:

Table 10-75 – CatchEvent attributes and model associations

Attribute Name	Description/Usage
eventDefinitionRefs: EventDefinition [0..*]	EventDefinitionRefs (EventDefinition) is an attribute that defines the type of reusable *triggers* expected for a *catch* Event. If there is no EventDefinition defined, then this is considered a *catch* None Event and the Event will not have an internal marker (see Figure 10-86). If there is more than one EventDefinition defined, this is considered a *Catch* Multiple Event and the Event will have the pentagon internal marker (see Figure 10-85).
eventDefinitions: EventDefinition [0..*]	EventDefinitionRefs (EventDefinition) is an attribute that defines the type of contained *triggers* expected for a *catch* Event. If there is no EventDefinition defined, then this is considered a *catch* None Event and the Event will not have an internal marker (see Figure 10-86). If there is more than one EventDefinition defined, this is considered a *Catch* Multiple Event and the Event will have the pentagon internal marker (see Figure 10-85).
dataOutputAssociations: DataOutputAssociation [0..*]	The Data Associations of the *catch* Event. The dataOutputAssociation of a *catch* Event is used to assign data from the Event to a data element that is in the scope of the Event. For a *catch* Multiple Event, multiple Data Associations might be required, depending on the individual *triggers* of the Event.
dataOutput: DataOutput [0..*]	The Data Outputs for the *catch* Event.
outputSet: OutputSet [0..1]	The OutputSet for the *catch* Event

Common Throw Event Attributes

The following table shows attributes that are common for Throw Events.

The ThrowEvent element inherits the attributes and model associations of Flow Element (see Table 8-45) through its relationship to the Event element (see page 103). Table 10-76 presents the additional attributes and model associations of the CatchEvent element.

Table 10-76 – ThrowEvent attributes and model associations

Attribute Name	**Description/Usage**
eventDefinitionRefs: EventDefinition [0..*]	EventDefinitionRefs (EventDefinition) is an attribute that defines the type of reusable *triggers* expected for a *throw* Event. If there is no EventDefinition defined, then this is considered a *throw* None Event and the Event will not have an internal marker (see Figure 10-86). If there is more than one EventDefinition defined, this is considered a *throw* Multiple Event and the Event will have the pentagon internal marker (see Figure 10-85).
eventDefinitions: EventDefinition [0..*]	EventDefinitionRefs (EventDefinition) is an attribute that defines the type of contained *results* expected for a *throw* Event. If there is no EventDefinition defined, this is considered a *throw* None Event and the Event will not have an Internal marker (see Figure 10-86). If there is more than one EventDefinition defined, this is considered a *throw* Multiple Event and the Event will have the pentagon internal marker (see Figure 10-85).
dataInputAssociations: DataInputAssociation [0..*]	The Data Associations of the *throw* Event. The dataInputAssociation of a *throw* Event is responsible for the assignment of a data element that is in scope of the Event to the Event data. For a *throw* Multiple Event, multiple data associations might be required, depending on the individual *results* of the Event.
dataInput: DataInput [0..*]	The Data Inputs for the *throw* Event.
inputSet: InputSet [0..1]	The InputSet for the *throw* Event

Implicit Throw Event

A sub-type of *throw* Event is the ImplicitThrowEvent. This is a non-graphical Event that this used for Multi-Instance Activities (see page 201). The ImplicitThrowEvent element inherits the attributes and model associations of ThrowEvent (see Table 10-76), but does not have any additional attributes or model associations.

10.4.2. Start Event

As the name implies, the Start Event indicates where a particular Process will start. In terms of Sequence Flow, the Start Event starts the flow of the Process, and thus, will not have any *incoming* Sequence Flow—no Sequence Flow can connect to a Start Event.

The Start Event shares the same basic shape of the Intermediate Event and End Event, a circle with an open center so that markers can be placed within the circle to indicate variations of the Event.

- A Start Event is a circle that MUST be drawn with a single thin line (see Figure 10-66).
    - The use of text, color, size, and lines for a Start Event MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63 with the exception that:
        - The thickness of the line MUST remain thin so that the Start Event may be distinguished from the Intermediate and End Events.

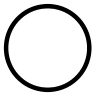


Figure 10-66 - Start Event

Throughout this document, we discuss how Sequence Flow is used within a Process. To facilitate this discussion, we employ the concept of a *token* that will traverse the Sequence Flow and pass through the elements in the Process. A *token* is a theoretical concept that is used as an aid to define the behavior of a Process that is being performed. The behavior of Process elements can be defined by describing how they interact with a *token* as it "traverses" the structure of the Process.

Note: A *token* does not traverse the Message Flow since it is a Message that is passed down a Message Flow (as the name implies).

Semantics of the Start Event include:

- A Start Event is OPTIONAL: a Process level—a top-level Process or an expanded Sub-Process—MAY (is no required to) have a Start Event.

Note – A Process may have more than one Process level (i.e., it can include Expanded Sub-Processes). The use of Start and End Events is independent for each level of the Diagram.

- If a Process is complex and/or the starting conditions are not obvious, then it is RECOMMENDED that a Start Event be used
- If a Start Event is not used, then the implicit Start Event for the Process SHALL NOT have a Trigger.
- If there is an End Event, then there MUST be at least one Start Event.
- If the Start Event is used, then there MUST NOT be other flow elements that do not have *incoming* Sequence Flow—all other Flow Objects MUST be a target of at least one Sequence Flow.
    - Exceptions to this are Activities that are defined as being Compensation Activities (have the Compensation Marker). Compensation Activities MUST NOT have any *incoming*

Sequence Flow, even if there is a Start Event in the Process level. See page 314 for more information on Compensation Activities.

  - ◆ An exception to this is the Intermediate Event, which MAY be without an *incoming* Sequence Flow (when attached to an Activity boundary).

- ◆ If the Start Event is not used, then all Flow Objects that do not have an *incoming* Sequence Flow (i.e., are not a target of a Sequence Flow) SHALL be instantiated when the Process is instantiated. There is an assumption that there is only one implicit Start Event, meaning that all the starting *Flow Objects* will start at the same time.

  - ◆ Exceptions to this are Activities that are defined as being Compensation Activities (have the Compensation marker). Compensation Activities are not considered a part of the *normal flow* and MUST NOT be instantiated when the Process is instantiated. See page 314 for more information on Compensation Activities.

- ◆ There MAY be multiple Start Events for a given Process level.

  - ◆ Each Start Event is an independent Event. That is, a Process *instance* SHALL be generated when the Start Event is triggered.

If the Process is used as a Sub-Process and there are multiple None Start Events, then when flow is transferred from the parent Process to the Sub-Process, only one of the Sub-Process's Start Events will be triggered. The TargetRef attribute of the Sequence Flow *incoming* to the Sub-Process object can be extended to identify the appropriate Start Event.

Note – The behavior of Process may be harder to understand if there are multiple Start Events. It is RECOMMENDED that this feature be used sparingly and that the modeler be aware that other readers of the Diagram may have difficulty understanding the intent of the Diagram.

When the trigger for a Start Event occurs, a new Process will be instantiated and a *token* will be generated for each *outgoing* Sequence Flow from that Event.

Start Event Triggers

Start Events can be used for three types of Processes:

- *Top-level* Processes
- Sub-Processes (*embedded* and *called* (*reusable*))
- Event Sub-Processes.

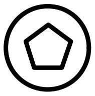
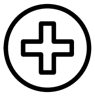
The next three (3) sections describe the types of Start Events that can be used for each of these three types of Processes.

Start Events for Top-level Processes

There are many ways that *top-level* Processes can be started (instantiated). The *Trigger* for a Start Event is designed to show the general mechanisms that will instantiate that particular Process. There are seven (7) types of Start Events for *top-level* Processes in BPMN (see Table 10-77): *None*, Message, Timer, Conditional, Signal, Multiple, and Parallel.

Table 10-77 – Top-Level Process Start Event Types

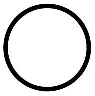
Trigger	Description	Marker
None	The `None Start Event` does not have a defined *trigger*. There is no specific `EventDefinition` subclass (see page 266) for `None Start Events`. If the `Start Event` has no associated `EventDefiniton`, then the Event MUST be displayed without a marker (see the figure on the right).	○
Message	A `Message` arrives from a *Participant* and triggers the start of the `Process`. See page 112 for more details on `Messages`. If there is only one (1) `EventDefinition` associated with the `Start Event` and that `EventDefinition` is of the subclass `MessageEventDefinition`, then the `Event` is a `Message Start Event` and MUST be displayed with an envelope marker (see the figure to the right). The actual *Participant* from which the `Message` is received can be identified by connecting the `Event` to a *Participant* using a `Message Flow` within the definitional `Collaboration` of the `Process` – see Table 10-1.	✉
Timer	A specific time-date or a specific cycle (e.g. every Monday at 9am) can be set that will trigger the start of the Process. If there is only one (1) `EventDefinition` associated with the `Start Event` and that `EventDefinition` is of the subclass `TimerEventDefinition`, then the `Event` is a Timer `Start Event` and MUST be displayed with a clock marker (see the figure to the right).	🕒
Conditional	This type of event is triggered when a Condition such as “S&P 500 changes by more than 10% since opening”, or “Temperature above 300C” become true. The Condition Expression for the Event must become false and then true before the Event can be triggered again. If there is only one (1) `EventDefinition` associated with the `Start Event` and that `EventDefinition` is of the subclass `ConditionalEventDefinition`, then the `Event` is a `Conditional Start Event` and MUST be displayed with a lined paper marker (see the figure to the right).	≣
Signal	A *Signal* arrives that has been broadcast from another `Process` and triggers the start of the `Process`. Note that the *Signal* is not a `Message`, which has a specific target for the `Message`. Multiple `Processes` can have `Start Events` that are triggered from the same broadcasted *Signal*. If there is only one (1) `EventDefinition` associated with the `Start Event` and that `EventDefinition` is of the subclass `SignalEventDefinition`, then the `Event` is a Signal `Start Event` and MUST be displayed with a triangle marker (see the figure to the right).	△

Multiple	This means that there are multiple ways of triggering the Process. Only one of them is required. There is no specific EventDefinition subclass (see page 266) for Multiple Start Events. If the Start Event has more than one associated EventDefiniton, then the Event MUST be displayed with the Multiple Event marker (a pentagon—see the upper figure to the right).	
Parallel Multiple	This means that there are multiple triggers required before the Process can be instantiated. All of the types of triggers that are listed in the Start Event MUST be triggered before the Process is instantiated. There is no specific EventDefinition subclass (see page 266) for Parallel Multiple Start Events. If the Start Event has more than one associated EventDefiniton and the parallelMultiple attribute of the Start Event is *true*, then the Event MUST be displayed with the Parallel Multiple Event marker (an open plus sign—see the figure to the right).	

Start Events for Sub-Processes

There is only one (1) type of Start Event for Sub-Processes in BPMN (see Table 10-78): None.

Table 10-78 – Sub-Process Start Event Types

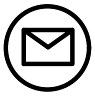
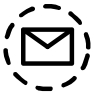
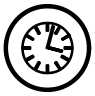
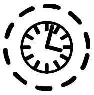
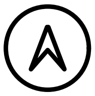
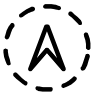
Trigger	Description	Marker
None	The None Start Event is used for all Sub-Processes, either *embedded* or *called* (reusable). Other types of *Triggers* are not used for a Sub-Process, since the flow of the Process (a token) from the *parent* Process is the *Trigger* of the Sub-Process. If the Sub-Process is called (reusable) and has multiple Start Events, some of the other Start Events may have *Triggers*, but these Start Events would not be used in the context of a Sub-Process. When the other Start Events are triggered, they would instantiate top-level Processes.	

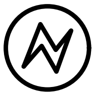
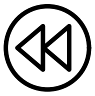
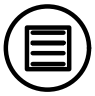
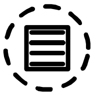
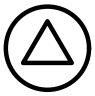
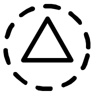
Start Events for Event Sub-Processes

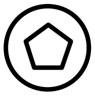
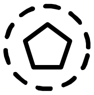
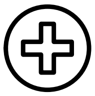
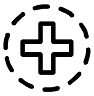
A Start Event can also initiate an inline Event Sub-Process (see page 188). In that case, the same Event types as for boundary Events are allowed (see Table 10-79), namely: Message, Timer, Escalation, Error, Cancel, Compensation, Conditional, Signal, Multiple, and Parallel.

◆ An Event Sub-Process MUST have a single Start Event.

Table 10-79 – Event Sub-Process Start Event Types

Trigger	Description	Marker
Message	If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass MessageEventDefinition, then the Event is a Message Start Event and uses an envelope marker (see the figures to the right). For a Message Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For a Message Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right). The actual *Participant* from which the Message is received can be identified by connecting the Event to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.	***Interrupting***  ***Non-Interrupting*** 
Timer	If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass TimerEventDefinition, then the Event is a Timer Start Event and uses a clock marker (see the figures to the right). For a Timer Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For a Timer Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 
Escalation	Escalation Event Sub-Processes implement measures to expedite the completion of a business activity, should it not satisfy a constraint specified on its execution (such as a time-based deadline). The Escalation Start Event is only allowed for triggering an in-line Event Sub-Process. If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass EscalationEventDefinition, then the Event is an Escalation Start Event and uses an arrowhead marker (see the figures to the right). For an Escalation Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For an Escalation Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 

Error	The Error Start Event is only allowed for triggering an in-line Event Sub-Process. If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass ErrorEventDefinition, then the Event is an Error Start Event and uses a lightning marker (see the figures to the right). Given the nature of Errors, an Event Sub-Process with an Error *trigger* will always interrupt its containing Process.	***Interrupting*** 
Compensation	The Compensation Start Event is only allowed for triggering an in-line Compensation Event Sub-Process (see “Compensation Handler” on page 314). This type of Event is triggered when *compensation* occurs. If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass CompensationEventDefinition, then the Event is a Compensation Start Event and uses a double triangle marker (see the figure to the right). This Event does not interrupt the Process since the Process has to be completed before this Event can be triggered.	
Conditional	If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass ConditionalEventDefinition, then the Event is a Conditional Start Event and uses an lined page marker (see the figures to the right). For a Conditional Event Sub-Process that interrupts its containing Process, then the boundary of the Event is solid (see the upper figure to the right). For a Conditional Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 
Signal	If there is only one (1) EventDefinition associated with the Start Event and that EventDefinition is of the subclass SignalEventDefinition, then the Event is a Signal Start Event and uses an triangle marker (see the figures to the right). For a Signal Event Sub-Process that interrupts its containing Process, then the boundary of the Event is solid (see the upper figure to the right). For a Signal Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 

Multiple	A Multiple Event indicates that that there are multiple ways of triggering the Event Sub-Process. Only one of them is required to actually start the Event Sub-Process. There is no specific EventDefinition subclass (see page 266) for Multiple Start Events. If the Start Event has more than one associated EventDefiniton, then the Event MUST be displayed with the Multiple Event marker (a pentagon—see the figures on the right). For a Multiple Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For a Multiple Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 
Parallel Multiple	A Multiple Event indicates that that there are multiple ways of triggering the Event Sub-Process. All of them are required to actually start the Event Sub-Process. There is no specific EventDefinition subclass (see page 266) for Parallel Multiple Start Events. If the Start Event has more than one associated EventDefiniton and the parallelMultiple attribute of the Start Event is *true*, then the Event MUST be displayed with the Parallel Multiple Event marker (an open plus sign—see the figures to the right). For a Parallel Multiple Event Sub-Process that interrupts its containing Process, the boundary of the Event is solid (see the upper figure to the right). For a Parallel Multiple Event Sub-Process that does not interrupt its containing Process, the boundary of the Event is dashed (see the lower figure on the right).	***Interrupting***  ***Non-Interrupting*** 

Attributes for Start Events

For Start Events, the following additional attribute exists:

The Start Event element inherits the attributes and model associations of CatchEvent (see Table 10-75). Table 10-80 presents the additional attributes of the Start Event element:

Table 10-80 – Start Event attributes

Attribute Name	Description/Usage
isInterrupting: boolean	This attribute only applies to Start Events of Event Sub-Processes; it is ignored for other Start Events. This attribute denotes whether the Sub-Process encompassing the Event Sub-Process should be cancelled or not, If the encompassing Sub-Process is not cancelled, multiple instances of the Event Sub-Process can run concurrently. This attribute cannot be applied to Error Events (where it's always *true*), or Compensation Events (where it doesn't apply).

Sequence Flow Connections

See Section "Sequence Flow Connections Rules" on page 64 for the entire set of objects and how they may be source or targets of Sequence Flow.

- A Start Event MUST NOT be a target for Sequence Flow; it MUST NOT have *incoming* Sequence Flow.
  - An exception to this is when a Start Event is used in an Expanded Sub-Process and is attached to the boundary of that Sub-Process. In this case, a Sequence Flow from the higher-level Process MAY connect to that Start Event in lieu of connecting to the actual boundary of the Sub-Process.
- A Start Event MUST be a source for Sequence Flow.
- Multiple Sequence Flow MAY originate from a Start Event. For each Sequence Flow that has the Start Event as a source, a new parallel path SHALL be generated.
  - The Condition attribute for all *outgoing* Sequence Flow MUST be set to None.
  - When a Start Event is not used, then all Flow Objects that do not have an *incoming* Sequence Flow SHALL be the start of a separate parallel path.
  - Each path will have a separate unique *token* that will traverse the Sequence Flow.

Message Flow Connections

Note – All Message Flow must connect two separate Pools. They can connect to the Pool boundary or to *Flow Objects* within the Pool boundary. They cannot connect two objects within the same Pool.

See Section "Message Flow Connection Rules" on page 65 for the entire set of objects and how they may be source or targets of Message Flow.

- A Start Event MAY be the target for Message Flow; it can have 0 (zero) or more *incoming* Message Flow. Each Message Flow arriving at a Start Event represents an instantiation mechanism (a Trigger) for the Process. Only one of the Triggers is required to start a new Process.

    - The Trigger attribute of the Start Event MUST be set to Message or Multiple if there are any *incoming* Message Flow.
    - The Trigger attribute of the Start Event MUST be set to Multiple if there are more than one *incoming* Message Flow.
- A Start Event MUST NOT be a source for Message Flow; it MUST NOT have *outgoing* Message Flow.

10.4.3. End Event

As the name implies, the End Event indicates where a Process will end. In terms of Sequence Flow, the End Event ends the flow of the Process, and thus, will not have any *outgoing* Sequence Flow—no Sequence Flow can connect from an End Event.

The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle with an open center so that markers can be placed within the circle to indicate variations of the Event.

- An End Event is a circle that MUST be drawn with a single thick line (see Figure 10-67).
- The use of text, color, size, and lines for an End Event MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63 with the exception that:
    - The thickness of the line MUST remain thick so that the End Event may be distinguished from the Intermediate and Start Events.

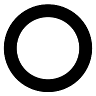


Figure 10-67 - End Event

To continue discussing how flow proceeds throughout the Process, an End Event consumes a *token* that had been generated from a Start Event within the same level of Process. If parallel Sequence Flow targets the End Event, then the *tokens* will be consumed as they arrive. All the *tokens* that were generated within the Process must be consumed by an End Event before the Process has been completed. In other circumstances, if the Process is a Sub-Process, it can be stopped prior to normal completion through interrupting Intermediate Events (See Section 10.2.2, "Exception Flow," on page 131 for more details). In this situation the *tokens* will be consumed by an Intermediate Event attached to the boundary of the Sub-Process.

Semantics of the End Event include:

- There MAY be multiple End Events within a single level of a Process.
- An End Event is OPTIONAL: a given Process level—a top-level Process or an expanded Sub-Process—MAY (is not required to) have this shape:
    - If an End Event is not used, then the implicit End Event for the Process SHALL NOT have a Result.
    - If there is a Start Event, then there MUST be at least one End Event.
    - If an End Event is used, then there MUST NOT be other flow elements that do not have any *outgoing* Sequence Flow—all other Flow Objects MUST be a source of at least one Sequence

Flow.

- Exceptions to this are Activities that are defined as being Compensation Activities (have the Compensation marker). Compensation Activities MUST NOT have any *outgoing* Sequence Flow, even if there is an End Event in the Process level. See page 314 for more information on Compensation Activities.
    - If the End Event is not used, then all Flow Objects that do not have any *outgoing* Sequence Flow (i.e., are not a source of a Sequence Flow) mark the end of a path in the Process. However, the Process MUST NOT end until all parallel paths have completed.
    - Exceptions to this are Activities that are defined as being Compensation Activities (have the Compensation marker). Compensation Activities are not considered a part of the *normal flow* and MUST NOT mark the end of the Process.

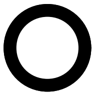
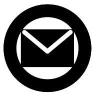
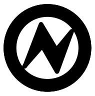
Note – A Process may have more than one Process level (i.e., it can include *Expanded* Sub-Processes). The use of Start and End Events is independent for each level of the Diagram.

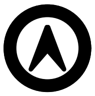
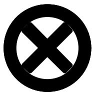
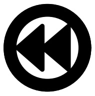
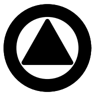
For Processes without an End Event, a *token* entering a path-ending Flow Object will be consumed when the processing performed by the object is completed (i.e., when the path has completed), as if the *token* had then gone on to reach an End Event. When all *tokens* for a given *instance* of the Process are consumed, then the Process will reach a state of being completed.

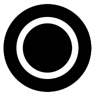
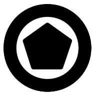
End Event Results

There are nine (9) types of End Events in BPMN: *None*, Message, Escalation, Error, Cancel, Compensation, Signal, Terminate, and Multiple. These types define the consequence of reaching an End Event. This will be referred to as the End Event *Result*.

Table 10-81 – End Event Types

Trigger	Description	Marker
None	The None End Event does not have a defined *result*. There is no specific EventDefinition subclass (see page 266) for None End Events. If the End Event has no associated EventDefiniton, then the Event will be displayed without a marker (see the figure on the right).	
Message	This type of End indicates that a Message is sent to a *Participant* at the conclusion of the Process. See page 112 for more details on Messages. The actual *Participant* from which the Message is received can be identified by connecting the Event to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.	
Error	This type of End indicates that a named Error should be generated. All currently active threads in the particular Sub-Process are terminated as a result. The Error will be caught by a *Catch* Error Intermediate Event with the same errorCode or no errorCode which is on the boundary of the nearest enclosing parent activity (hierarchically). The behavior of the	

	Process is unspecified if no Activity in the hierarchy has such an Error Intermediate Event. The system executing the process may define additional *Error* handling in this case, a common one being termination of the Process *instance*.	
Escalation	This type of End indicates that an *Escalation* should be triggered. Other active threads are not affected by this and continue to be executed. The *Escalation* will be caught by a *Catch* Escalation Intermediate Event with the same escalationCode or no escalationCode which is on the boundary of the nearest enclosing parent activity (hierarchically). The behavior of the process is unspecified if no activity in the hierarchy has such an Escalation Intermediate Event.	
Cancel	This type of End is used within a Transaction Sub-Process. It will indicate that the Transaction should be cancelled and will trigger a Cancel Intermediate Event attached to the Sub-Process boundary. In addition, it will indicate that a Transaction Protocol Cancel message should be sent to any Entities involved in the Transaction.	
Compensation	This type of End indicates that *compensation* is necessary. If an Activity is identified, and it was successfully completed, then that Activity will be compensated. The Activity must be visible from the Compensation End Event, i.e., one of the following must be true: • The Compensation End Event is contained in *normal flow* at the same level of Sub-Process. • The Compensation End Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activity. If no Activity is identified, all successfully completed Activities visible from the Compensation End Event are compensated, in reverse order of their Sequence Flow. Visible means one of the following: • The Compensation End Event is contained in *normal flow* and at the same level of Sub-Process as the Activities. • The Compensation End Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activities. To be compensated, an Activity MUST have a boundary Compensation Event or contain a Compensation Event Sub-Process.	
Signal	This type of End indicates that a Signal will be broadcasted when the End has been reached. Note that the Signal, which is broadcast to any Process that can receive the Signal, can be sent across Process levels or Pools, but is not a Message (which has a specific Source and Target). The attributes of a Signal can be found on page 282.	

Terminate	This type of End indicates that all activities in the Process should be immediately ended. This includes all instances of Multi-Instances. The Process is ended without *compensation* or *event handling*.	
Multiple	This means that there are multiple consequences of ending the Process. All of them will occur (e.g., there might be multiple messages sent). There is no specific EventDefinition subclass (see page 266) for Multiple End Events. If the End Event has more than one associated EventDefiniton, then the Event will be displayed with the Multiple Event marker (a pentagon—see the figure on the right).	

Sequence Flow Connections

See Section “Sequence Flow Connections Rules” on page 64 for the entire set of objects and how they may be source or targets of Sequence Flow.

- An End Event MUST be a target for Sequence Flow.
- An End Event MAY have multiple *incoming* Sequence Flow.

The Flow MAY come from either alternative or parallel paths. For modeling convenience, each path MAY connect to a separate End Event object. The End Event is used as a Sink for all *tokens* that arrive at the Event. All *tokens* that are generated at the Start Event for that Process must eventually arrive at an End Event. The Process will be in a running state until all *tokens* are consumed.

- An End Event MUST NOT be a source for Sequence Flow; that is, there MUST NOT be *outgoing* Sequence Flow.
    - An exception to this is when an End Event is used in an Expanded Sub-Process and is attached to the boundary of that Sub-Process. In this case, a Sequence Flow from the higher-level Process MAY connect from that End Event in lieu of connecting from the actual boundary of the Sub-Process (see **[-->REF]**).

Message Flow Connections

See Section “Message Flow Connection Rules” on page 65 for the entire set of objects and how they may be source or targets of Message Flow.

Note – All Message Flow must connect two separate Pools. They can connect to the Pool boundary or to *Flow Objects* within the Pool boundary. They cannot connect two objects within the same Pool.

- An End Event MUST NOT be the target for Message Flow; it can have no *incoming* Message Flow. If the Intermediate Event has an *incoming* Message Flow, then it MUST NOT have an *outgoing* Message Flow.

- An Intermediate Event of type Message, if it is used within *normal flow*, MAY be the source for Message Flow; it can have one *outgoing* Message Flow. If the Intermediate Event has an *outgoing* Message Flow, then it MUST NOT have an *incoming* Message Flow.

10.4.4. Intermediate Event

As the name implies, the Intermediate Event indicates where something happens (an Event) somewhere between the start and end of a Process. It will affect the flow of the Process, but will not start or (directly) terminate the Process. Intermediate Events can be used to:

- Show where Messages are expected or sent within the Process,
- Show delays are expected within the Process,
- Disrupt the *normal flow* through *exception handling*, or
- Show the extra work required for *compensation*.

The Intermediate Event shares the same basic shape of the Start Event and End Event, a circle with an open center so that markers can be placed within the circle to indicate variations of the Event.

- An Intermediate Event is a circle that MUST be drawn with a double thin line. (see Figure 10-68).
  - The use of text, color, size, and lines for an Intermediate Event MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63 with the exception that:
    - The thickness of the line MUST remain double so that the Intermediate Event may be distinguished from the Start and End Events.

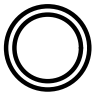


Figure 10-68 – Intermediate Event

One use of Intermediate Events is to represent exception or *compensation handling*. This will be shown by placing the Intermediate Event on the boundary of a Task or Sub-Process (either collapsed or expanded). The Intermediate Event can be attached to any location of the Activity boundary and the *outgoing* Sequence Flow can flow in any direction. However, in the interest of clarity of the Diagram, we recommend that the modeler choose a consistent location on the boundary. For example, if the Diagram orientation is horizontal, then the Intermediate Events can be attached to the bottom of the Activity and the Sequence Flow directed down, then to the right. If the Diagram orientation is vertical, then the Intermediate Events can be attached to the left or right side of the Activity and the Sequence Flow directed to the left or right, then down.

Intermediate Event Triggers

There are twelve (12) types of Intermediate Events in BPMN: *None*, Message, Timer, Escalation, Error, Cancel, Compensation, Conditional, Link, Signal, Multiple, and Parallel Multiple. Each type of Intermediate Event will have a different icon placed in the center of the Intermediate Event shape to distinguish one from another.

There are two (2) ways that Intermediate Events are used in BPMN:

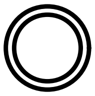
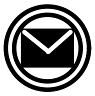
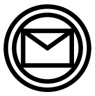
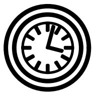
An Intermediate Event that is placed within the *normal flow* of a Process can be used for one of two purposes. The Event can respond to ("catch") the Event Trigger or the Event can be used to set off ("throw") the Event Trigger. An Intermediate Event that is attached to the boundary of an Activity can only be used to “catch” the Event Trigger.

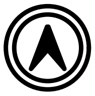
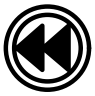
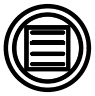
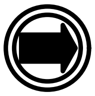
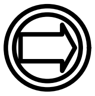
Intermediate Events in Normal Flow

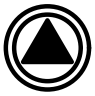
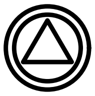
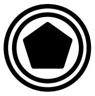
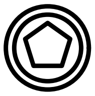
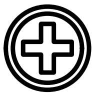
When a *token* arrives at an Intermediate Event that is placed within the *normal flow* of a Process, one of two things will happen. If the Event is used to “throw” the Event Trigger, then Trigger of the Event will immediately occur (e.g., the Message will be sent) and the *token* will move down the *outgoing* Sequence Flow. If the Event is used to “catch” the Event Trigger, then the *token* will remain at the Event until the Trigger occurs (e.g., the Message is received). Then the *token* will move down the *outgoing* Sequence Flow.

Nine (9) of the eleven (11) Intermediate Events can be used in *Normal Flow.* Table 10-82

Table 10-82 – Intermediate Event Types in Normal Flow

Trigger	Description	Marker
None	The None Intermediate Event is only valid in *Normal Flow*, i.e. it may not be used on the boundary of an Activity. Although there is no specific trigger for this Event, it is defined as *throw* Event. It is used for modeling methodologies that use Events to indicate some change of state in the Process. There is no specific EventDefinition subclass (see page 266) for None Intermediate Events. If the (throw) Intermediate Event has no associated EventDefiniton, then the Event MUST be displayed without a marker (see the figure on the right).	***Throw*** 
Message	A Message Intermediate Event can be used to either send a Message or receive a Message. When used to “throw” the message, the Event marker MUST be filled (see the upper figure on the right). When used to “catch” the message, then the Event marker MUST be unfilled (see the lower figure on the right). This causes the Process to continue if it was waiting for the message, or changes the flow for exception handling. The actual *Participant* from which the Message is received can be identified by connecting the Event to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1. See page 112 for more details on Messages.	***Throw***  ***Catch*** 
Timer	In *Normal Flow* the Timer Intermediate Event acts as a delay mechanism based on a specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the Event. This Event MUST be displayed with a clock marker (see the figure on the right).	***Catch*** 

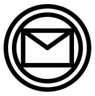
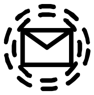
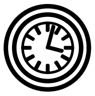
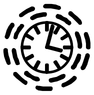
Escalation	In *Normal Flow*, the Escalation Intermediate Event *raises* an *Escalation*. Since this is a *Throw* Event, the arrowhead marker will be filled (see the figure to the right).	***Throw*** 
Compensation	In *normal flow*, this Intermediate Event indicates that *compensation* is necessary. Thus, it is used to "throw" the Compensation Event, and the Event marker MUST be filled (see figure on the right). If an Activity is identified, and it was successfully completed, then that Activity will be compensated. The activity must be visible from the Compensation Intermediate Event, i.e., one of the following must be true: • The Compensation Intermediate Event is contained in *normal flow* at the same level of Sub-Process. • The Compensation Intermediate Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activity. If no Activity is identified, all successfully completed Activities visible from the Compensation Intermediate Event are compensated, in reverse order of their Sequence Flow. Visible means one of the following: • The Compensation Intermediate Event is contained in *normal flow* and at the same level of Sub-Process as the Activities. • The Compensation Intermediate Event is contained in a Compensation Event Sub-Process which is contained in the Sub-Process containing the Activities. To be compensated, an Activity MUST have a boundary Compensation Event, or contain a Compensation Event Sub-Process.	***Throw*** 
Conditional	This type of event is triggered when a *Condition* becomes true. A *Condition* is a type of `Expression`. The attributes of an `Expression` can be found page 106.	***Catch*** 
Link	The Link Intermediate Events are only valid in *Normal Flow*, i.e. they may not be used on the boundary of an Activity. A Link is a mechanism for connecting two sections of a Process. Link Events can be used to create looping situations or to avoid long Sequence Flow lines. Link Event uses are limited to a single Process level (i.e., they cannot link a parent Process with a Sub-Process). Paired Intermediate Events can also be used as “Off-Page Connectors” for printing a Process across multiple pages. They can also be used as generic “Go To” objects within the Process level. There can be multiple Source Link Events, but there can only be one Target Link Event. When used to “throw” to the Target Link, the Event marker will be filled (see the top figure on the right). When used to “catch” from the Source Link, the Event marker will be unfilled (see the bottom figure on the right).	***Throw***  ***Catch*** 

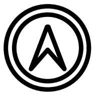
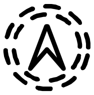
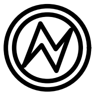
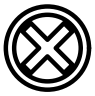
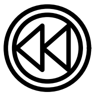
Signal	This type of event is used for sending or receiving Signals. A Signal is for general communication within and across Process Levels, across Pools, and between Business Process Diagrams. A BPMN Signal is similar to a signal flare that shot into the sky for anyone who might be interested to notice and then react. Thus, there is a source of the Signal, but no specific intended target. This type of Intermediate Event can send or receive a Signal if the Event is part of a Normal Flow. The Event can only receive a Signal when attached to the boundary of an activity. The Signal Event differs from an Error Event in that the Signal defines a more general, non-error condition for interrupting activities (such as the successful completion of another activity) as well as having a larger scope than Error Events. When used to "catch" the signal, the Event marker will be unfilled (see the middle figure on the right). When used to "throw" the signal, the Event marker will be filled (see the top figure on the right). The attributes of a Signal can be found on page 280.	***Throw***  ***Catch*** 
Multiple	This means that there are multiple Triggers assigned to the Event. If used within normal flow, the Event can "catch" the Trigger or "throw" the Triggers. When attached to the boundary of an activity, the Event can only "catch" the Trigger. When used to "catch" the Trigger, only one of the assigned Triggers is required and the Event marker will be unfilled (see the middle figure on the right). When used to "throw" the Trigger (the same as a Multiple End Event), all the assigned Triggers will be thrown and the Event marker will be filled (see the top figure on the right). There is no specific EventDefinition subclass (see page 266) for Multiple Intermediate Events. If the Intermediate Event has more than one associated EventDefiniton, then the Event will be displayed with the Multiple Event marker.	***Throw***  ***Catch*** 
Parallel Multiple	This means that there are multiple *triggers* assigned to the Event. If used within normal flow, the Event can only "catch" the *trigger*. When attached to the boundary of an activity, the Event can only "catch" the *trigger*. Unlike the normal Multiple Intermediate Event, all of the assigned *triggers* are required for the Event to be triggered. The Event marker will be an unfilled plus sign (see the figure on the right). There is no specific EventDefinition subclass (see page 266) for Parallel Multiple Intermediate Events. If the Intermediate Event has more than one associated EventDefiniton and the parallelMultiple attribute of the Intermediate Event is *true*, then the Event will be displayed with the Parallel Multiple Event marker.	

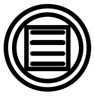
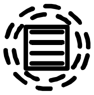
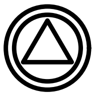
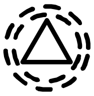
Intermediate Events Attached to an Activity Boundary

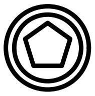
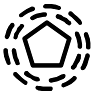
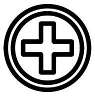
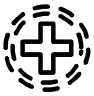
Table 10-83 describes the Intermediate Events that can be attached to the boundary of an Activity.

Table 10-83 – Intermediate Event Types Attached to an Activity Boundary

Trigger	Description	Marker
Message	A Message arrives from a participant and triggers the Event. If a Message Event is attached to the boundary of an Activity, it will change the *Normal Flow* into an *Exception Flow* upon being triggered. For a Message Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a Message Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*. The actual *Participant* from which the Message is received can be identified by connecting the Event to a *Participant* using a Message Flow within the definitional Collaboration of the Process – see Table 10-1.	***Interrupting***  ***Non-Interrupting*** 
Timer	A specific time-date or a specific cycle (e.g., every Monday at 9am) can be set that will trigger the Event. If a Timer Event is attached to the boundary of an Activity, it will change the Normal Flow into an Exception Flow upon being triggered. For a Timer Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a Timer Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 

Escalation	This type of Event is used for handling a named Escalation. If attached to the boundary of an activity, the Intermediate Event *catches* an Escalation. In contrast to an Error, an Escalation by default is assumed to not abort the activity to which the boundary event is attached. However, a modeler may decide to override this setting by using the notation described in the following. For an Escalation Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For an Escalation Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 
Error	An Intermediate Error Catch Event can only be attached to the boundary of an activity, i.e. it may not be used in Normal Flow. If used in this context, it reacts to (catches) a named error, or to any error if a name is not specified. Note that an Error Event always interrupts the Activity to which it is attached, i.e. there is not a non-interrupting version of this Event. The boundary of the Event thus always solid (see figure on the right).	***Interrupting*** 
Cancel	This type of Intermediate Event is used within a Transaction Sub-Process. This type of Event MUST be attached to the boundary of a Sub-Process. It SHALL be triggered if a Cancel End Event is reached within the Transaction Sub-Process. It also SHALL be triggered if a Transaction Protocol "Cancel" message has been received while the Transaction is being performed. Note that a Cancel Event always interrupts the Activity to which it is attached, i.e. there is not a non-interrupting version of this Event. The boundary of the Event thus always solid (see figure on the right).	***Interrupting*** 
Compensation	When attached to the boundary of an Activity, this Event is used to "catch" the Compensation Event, thus the Event marker MUST be unfilled (see figure on the right). The Event will be triggered by a thrown *compensation* targeting that Activity. When the Event is triggered, the Compensation Activity that is Associated to the Event will be performed (see page 314). Note that the interrupting a non-interrupting aspect of other Events does not apply in the case of a Compensation Event. *Compensations* can only be triggered after completion of the activity to which they are attached. Thus they cannot interrupt the Activity. The boundary of the Event is always solid.	

Conditional	This type of event is triggered when a *Condition* becomes true. A *Condition* is a type of `Expression`. The attributes of an `Expression` can be found page 106. If a Conditional Event is attached to the boundary of an Activity, it will change the *normal flow* into an *exception flow* upon being triggered. For a `Conditional Event` that interrupts the Activity to which it is attached, the boundary of the `Event` is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a `Conditional Event` that does not interrupt the Activity to which it is attached, the boundary of the `Event` is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 
Signal	The Signal Event can only receive a Signal when attached to the boundary of an activity. In this context, it will change the Normal Flow into an Exception Flow upon being triggered. The Signal Event differs from an Error Event in that the Signal defines a more general, non-error condition for interrupting activities (such as the successful completion of another activity) as well as having a larger scope than Error Events. When used to “catch” the signal, the Event marker will be unfilled. The attributes of a Signal can be found on page 280. For a `Signal Event` that interrupts the Activity to which it is attached, the boundary of the `Event` is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a `Signal Event` that does not interrupt the Activity to which it is attached, the boundary of the `Event` is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 

Multiple	A Multiple Event indicates that there are multiple Triggers assigned to the Event. When attached to the boundary of an activity, the Event can only “catch” the Trigger. In this case, only one of the assigned Triggers is required and the Event marker will be unfilled Upon being triggered, the Event that occurred will change the Normal Flow into an Exception Flow. There is no specific EventDefinition subclass (see page 266) for Multiple Intermediate Events. If the Intermediate Event has more than one associated EventDefiniton, then the Event will be displayed with the Multiple Event marker. For a Multiple Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see upper figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a Multiple Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see lower figure on the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 
Parallel Multiple	This means that there are multiple *triggers* assigned to the Event. When attached to the boundary of an activity, the Event can only “catch” the *trigger*. Unlike the normal Multiple Intermediate Event, <u>all</u> of the assigned *triggers* are required for the Event to be triggered. The Event marker will be an unfilled plus sign (see the figures on the right). There is no specific EventDefinition subclass (see page 266) for Parallel Multiple Intermediate Events. If the Intermediate Event has more than one associated EventDefiniton <u>and</u> the parallelMultiple attribute of the Intermediate Event is *true*, then the Event will be displayed with the Parallel Multiple Event marker. For a Parallel Multiple Event that interrupts the Activity to which it is attached, the boundary of the Event is solid (see the upper figure to the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *true*. For a Parallel Multiple Event that does not interrupt the Activity to which it is attached, the boundary of the Event is dashed (see the lower figure to the right). Note that if using this notation, the attribute cancel Activity of the Activity to which the Event is attached is implicitly set to *false*.	***Interrupting***  ***Non-Interrupting*** 

Attributes for Boundary Events

For boundary Events, the following additional attributes exists:

The BoundaryEvent element inherits the attributes and model associations of CatchEvent (see Table 8-45). Table 8-47 presents the additional attributes and model associations of the Boundary Event element:

Table 10-84 – Boundary Event attributes

Attribute Name	Description/Usage
attachedTo: Activity	Denotes the Activity that boundary Event is attached to.
cancelActivity: boolean	Denotes whether the activity should be cancelled or not, i.e., whether the boundary catch event acts as an error or an escalation. If the activity is not cancelled, multiple instances of that handler can run concurrently. This attribute cannot be applied to error events (where it's always true), or Compensation Events (where it doesn't apply).

The following table specifies whether the cancel Activity attribute can be set on a boundary Event depending on the EventDefinition it catches.

Table 10-85 – Possible Values of the cancel Activity Attribute

Trigger	Possible Values for the cancel Activity Attribute
None	N/A as this event cannot be attached to the activity border.
Message	True/False
Timer	True/False
Escalation	True/False
Error	True
Cancel	True
Compensation	N/A as the scope was already executed and can no longer be canceled when *compensation* is triggered.
Conditional	True/False

Signal	True/False
Multiple	True/False if all Event Triggers allow this option (see this table for details). Otherwise the more restrictive option, i.e. Yes in case any Error or Cancel triggers are used.

Activity Boundary Connections

An Intermediate Event can be attached to the boundary of an Activity under the following conditions:

- (One or more) Intermediate Events MAY be attached directly to the boundary of an Activity.
  - To be attached to the boundary of an Activity, an Intermediate Event MUST be one of the following Triggers (EventDefinition): Message, Timer, Error, Escalation, Cancel, Compensation, Conditional, Signal, Multiple, and Parallel Multiple.
    - An Intermediate Event with a Cancel *Trigger* MAY be attached to a Sub-Process boundary only if the Transaction attribute of the Sub-Process is set to *true*.

Sequence Flow Connections

See Section "Sequence Flow Connections Rules" on page 64 for the entire set of objects and how they may be source or targets of Sequence Flow.

- The Intermediate Events with the following *Triggers* (EventDefinition) MAY be attached to the boundary of an Activity: Message, Timer, Error, Cancel (only Sub-Process that is a *Transaction*), Compensation, Conditional, Signal, Multiple, and Parallel Multiple. Thus, the following MUST NOT: None, and Link.
  - If the Intermediate Event is attached to the boundary of an Activity:
    - The Intermediate Event MUST NOT be a target for Sequence Flow; it cannot have an *incoming* Flow.
    - The Intermediate Event MUST be a source for Sequence Flow.
    - Multiple Sequence Flow MAY originate from an Intermediate Event. For each Sequence Flow that has the Intermediate Event as a source, a new parallel path SHALL be generated.
      - An exception to this: an Intermediate Event with a Compensation *Trigger* MUST NOT have an *outgoing* Sequence Flow (it MAY have an *outgoing* Association).
- The Intermediate Events with the following *Triggers* (EventDefinition) MAY be used in *normal flow*: None, Message, Timer, Compensation, Conditional, Link, and Signal. Thus, the following MUST NOT: Cancel, Error, Multiple, and Parallel Multiple.
  - If the Intermediate Event is used within *normal flow*:
    - Intermediate Events MUST be a target of a Sequence Flow.

Note – this is a change from BPMN 1.2 semantics, which allowed some Intermediate Events to not have an *incoming* Sequence Flow.

        - An Intermediate Event MAY have multiple *incoming* Sequence Flow.
            - **Note** – If the Event has multiple *incoming* Sequence Flow, then this is considered *uncontrolled flow*. This means that when a *token* arrives from one of the Paths, the Event will be enabled (to *catch* or *throw*). It will not wait for the arrival of *tokens* from the other paths. If another *token* arrives from the same path or another path, then a separate instance of the Event will be created. If the flow needs to be controlled, then the flow should converge with a Gateway that precedes the Event (see page 295 for more information on Gateways).

- An Intermediate Event MUST be a source for Sequence Flow.
- Multiple Sequence Flow MAY originate from an Intermediate Event. For each Sequence Flow that has the Intermediate Event as a source, a new parallel path SHALL be generated.
    - An exception to this: a *source* Link Intermediate Event (as defined below), it is not required to have an *outgoing* Sequence Flow.
- A Link Intermediate Event MUST NOT be both a *target* and a *source* of a Sequence Flow.

To define the use of a Link Intermediate Event as an "Off-Page Connector" or a "Go To" object:

- A Link Intermediate Event MAY be the target (*target* Link) or a source (*source* Link) of a Sequence Flow, but MUST NOT be both a *target* and a *source*.
    - If there is a *source* Link, there MUST be a matching *target* Link (they have the same name).
        - There MAY be multiple *source* Links for a single *target* Link.
        - There MUST NOT be multiple *target* Links for a single *source* Link.

Message Flow Connections

See Section "Message Flow Connection Rules" on page 65 for the entire set of objects and how they may be source or targets of Message Flow.

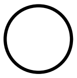
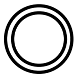
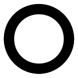
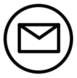
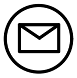
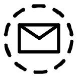
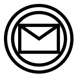
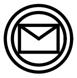
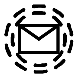
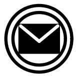
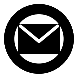
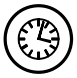
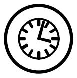
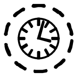
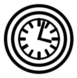
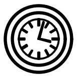
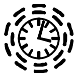
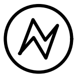
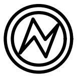
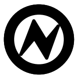
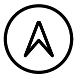
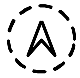
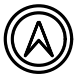
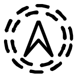
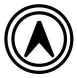
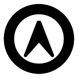
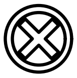
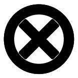
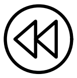
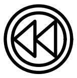
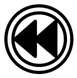
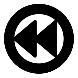
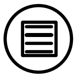
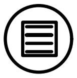
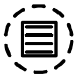
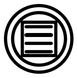
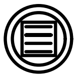
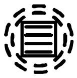
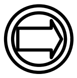
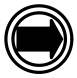
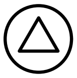
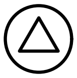
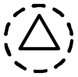
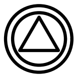
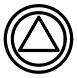
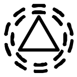
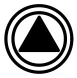
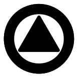
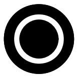
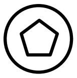
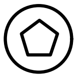
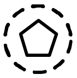
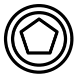
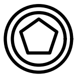
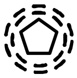
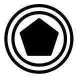
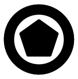
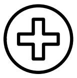
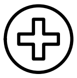
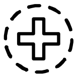
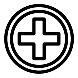
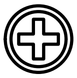
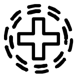
Note – All Message Flow must connect two separate Pools. They can connect to the Pool boundary or to Flow Objects within the Pool boundary. They cannot connect two objects within the same Pool.

- A Message Intermediate Event MAY be the *target* for Message Flow; it can have one *incoming* Message Flow.
- A Message Intermediate Event MAY be a *source* for Message Flow; it can have one *outgoing* Message Flow.
- A Message Intermediate Event MAY have an *incoming* Message Flow or an *outgoing* Message Flow, but not both.

10.4.5. Event Definitions

Event Definitions refers to the Triggers of *Catch* Events (Start and *receive* Intermediate Events) and the Results of *Throw* Events (End Events and *send* Intermediate Events). The types of Event Definitions are: CancelEventDefinition, CompensationEventDefinition, ConditionalEventDefinition, ErrorEventDefinition, EscalationEventDefinition, MessageEventDefinition, LinkEventDefinition, SignalEventDefinition, TerminateEventDefinition, and TimerEventDefinition (see Table 10-86). A None Event is determined by an Event that does not specify an Event Definition. A Multiple Event is determined by an Event that specifies more than one Event Definition. The different types of Events (Start, End, and Intermediate) utilize a subset of the available types of Event Definitions.

Table 10-86 – Types of Events and their Markers

Types	**Start**			**Intermediate**				**End**
	Top-Level	Event Sub-Process *Interrupting*	Event Sub-Process *Non-Interrupting*	Catching	Boundary *Interrupting*	Boundary *Non-Interrupting*	Throwing	
None								
Message								
Timer								
Error								
Escalation								
Cancel								
Compensation								
Conditional								
Link								
Signal								
Terminate								
Multiple								
Parallel Multiple								

The following sections will present the attributes common to all Event Definitions and the specific attributes for the Event Definitions that have additional attributes. Note that the Cancel and Terminate Event Definitions do not have additional attributes.

Event Definition Metamodel

Figure 10-69 shows the class diagram for the abstract class EventDefinition. When one of the EventDefinition sub-types (e.g., TimerEventDefinition) is defined it is contained in Definitions.

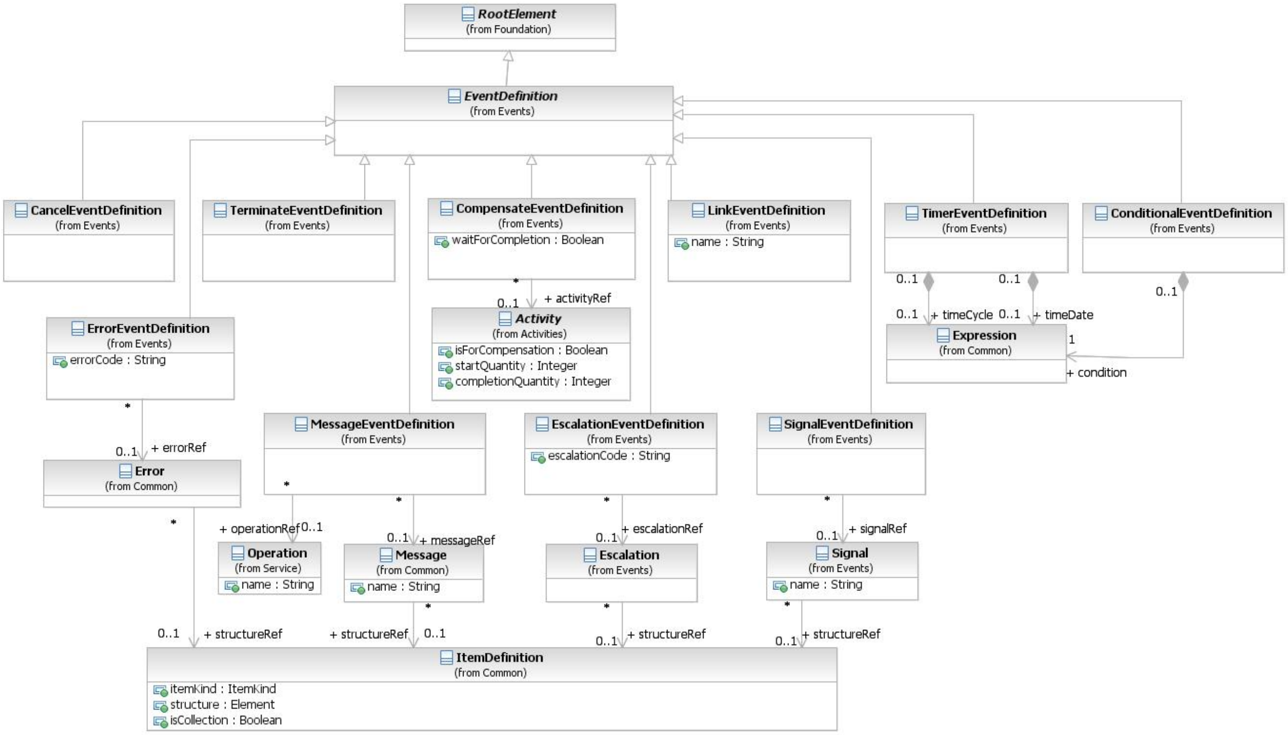


Figure 10-69 – EventDefinition Class Diagram

The EventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to RootElement, but does not contain any additional attributes or model associations.

The ErrorEventDefinition, EscalationEventDefinition and SignalEventDefinition subclasses comprise of attributes to carry data. The data is defined as part of the Events package. The MessageEventDefinition subclass comprises of an attribute that refers to a Message which is defined as part of the Collaboration package.

The following sections will present the sub-types of EventDefinitions.

Cancel Event

Cancel Events are only used in the context of modeling Transaction Sub-Processes (see page 188 for more details on *Transactions*). There are two (2) variations: a *catch* Intermediate Event and an End Event (Figure 10-70).

- The *catch* Cancel Intermediate Event MUST only be attached to the boundary of a Transaction Sub-Process and, thus, MAY NOT be used in *Normal Flow*.
- The Cancel End Event MUST only be used within a Transaction Sub-Process and, thus, MAY NOT be used in any other type of Sub-Process or Process.

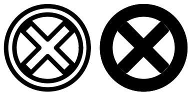


Figure 10-70 – Cancel Events

The CancelEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269).

Compensation Event

Compensation Events are used in the context of triggering or handling *compensation* (see page 314 for more details on *compensation*). There are four (4) variations: a Start Event, both a *catch* and *throw* Intermediate Event, and an End Event (Figure 10-71).

- The Compensation Start Event MAY NOT be used for a *top-level* Process.
- The Compensation Start Event MAY be used for an Event Sub-Process.
- The *catch* Compensation Intermediate Event MUST only be attached to the boundary of an Activity and, thus, MAY NOT be used in *Normal Flow*.
- The *throw* Compensation Intermediate Event MAY be used in *Normal Flow*.
- The Compensation End Event MAY be used within any Sub-Process or Process.

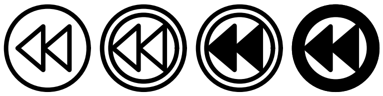


Figure 10-71 – Compensation Events

Figure 10-72 displays the class diagram for the CompensationEventDefinition.

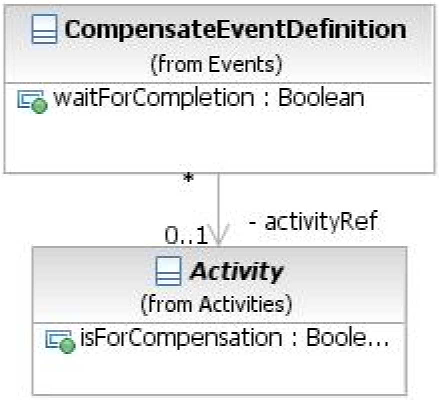

CompensateEventDefinition
(from Events)
waitForCompletion : Boolean
*
- activityRef
0..1
Activity
(from Activities)
isForCompensation : Boole...

Figure 10-72 – CompensationEventDefinition Class Diagram

The CompensationEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269). Table 10-87 presents the additional attributes and model associations of the CompensationEventDefinition element:

Table 10-87 – CompensationEventDefinition attributes and model associations

Attribute Name	Description/Usage
activityRef: Activity [0..1]	For a Start Event: This Event "catches" the *compensation* for an Event Sub-Process. No further information is required. The Event Sub-Process will provide the Id necessary to match the Compensation Event with the Event that *threw* the *compensation*, or the *compensation* will have been a broadcast. For an End Event: The Activity to be compensated MAY be supplied. If an Activity is not supplied, then the *compensation* is broadcast to all completed Activities in the current Sub-Process (if present), or the entire Process *Instance* (if at the global level). For an Intermediate Event within *Normal Flow*: The Activity to be compensated MAY be supplied. If an Activity is not supplied, then the *compensation* is broadcast to all completed Activities in the current Sub-Process (if present), or the entire Process *Instance* (if at the global level). This "throws" the *compensation*. For an Intermediate Event attached to the boundary of an Activity: This Event "catches" the *compensation*. No further information is required. The Activity the Event is attached to will provide the Id necessary to match the Compensation Event with the Event that *threw* the *compensation*, or the *compensation* will have been a broadcast.
waitForCompletion: boolean = True	For a *throw* Compensation Event, this flag determines whether the *throw* Intermediate Event waits for the triggered *compensation* to complete (the default), or just triggers the *compensation* and immediately continues (the BPMN 1.2 behavior).

Conditional Event

Figure 10-73 displays the class diagram for the ConditionalEventDefinition.

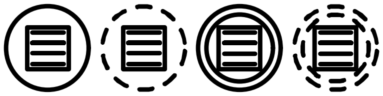


Figure 10-73 – Conditional Events

The ConditionalEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269). Table 10-88 presents the additional model associations of the ConditionalEventDefinition element:

Table 10-88 – ConditionalEventDefinition model associations

Attribute Name	Description/Usage
condition: Expression	The Expression might be underspecified and provided in the form of natural language. For executable Processes (processType = executable), if the *trigger* is Conditional, then a FormalExpression MUST be entered.

Error Event

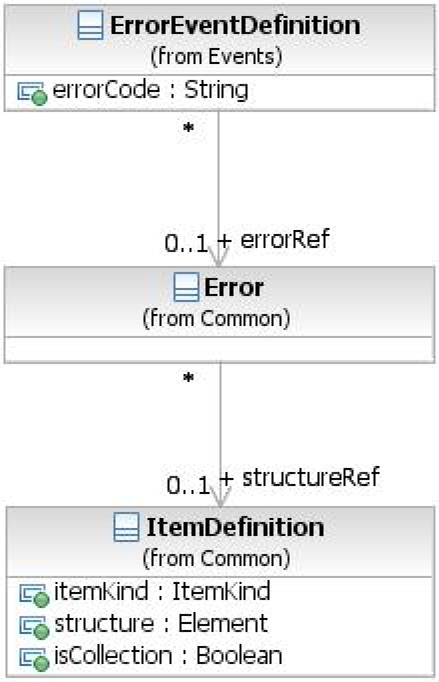


Figure 10-74 – ErrorEventDefinition Class Diagram

Figure 10-75

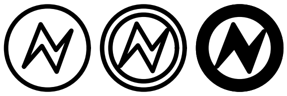


Figure 10-75 – Error Events

The `ErrorEventDefinition` element inherits the attributes and model associations of `BaseElement` (see Table 8-5) through its relationship to the `EventDefinition` element (see page 269). Table 10-89 presents the additional attributes and model associations of the `ErrorEventDefinition` element:

Table 10-89 – ErrorEventDefinition attributes and model associations

Attribute Name	Description/Usage
errorCode: string	For an End Event: If the Result is an Error, then the `errorCode` MUST be supplied (if the `processType` attribute of the `Process` is set to `executable`) This "throws" the error. For an Intermediate Event within Normal Flow: If the Trigger is an Error, then the `errorCode` MUST be entered (if the `processType` attribute of the `Process` is set to `executable`). This "throws" the Error. For an Intermediate Event attached to the boundary of an Activity: If the Trigger is an Error, then the `errorCode` MAY be entered. This Event "catches" the error. If there is no `errorCode`, then any error SHALL trigger the Event. If there is an `errorCode`, then only an error that matches the `errorCode` SHALL trigger the Event.
error: Error [0..1]	If the *Trigger* is an *Error*, then an *Error* payload MAY be provided.

Escalation Event Definition

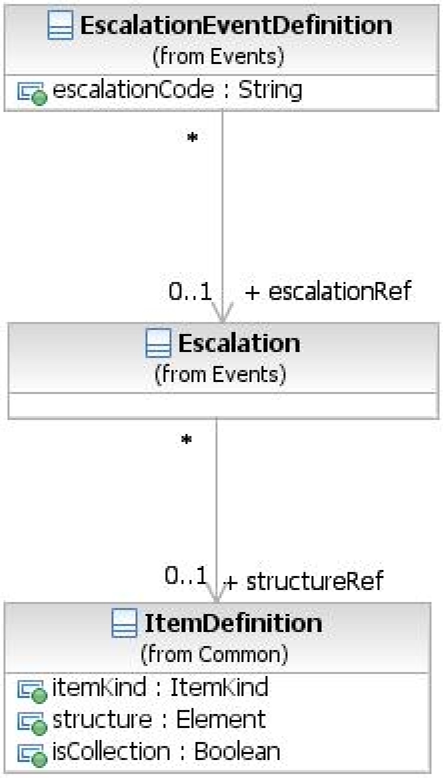


Figure 10-76 – EscalationEventDefinition Class Diagram

Figure 10-77

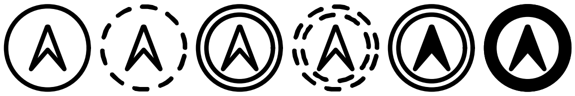


Figure 10-77 – Escalation Events

The EscalationEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269). Table 10-90 presents the additional attributes and model associations of the EscalationEventDefinition element:

Table 10-90 – EscalationEventDefinition attributes and model associations

Attribute Name	Description/Usage
escalationCode: string	For an End Event: If the Result is an Escalation, then the escalationCode MUST be supplied (if the processType attribute of the Process is set to executable). This "throws" the escalation. For an Intermediate Event within Normal Flow: If the Trigger is an Escalation, then the escalationCode MUST be entered (if the processType attribute of the Process is set to executable). This "throws" the escalation. For an Intermediate Event attached to the boundary of an Activity: If the Trigger is an Escalation, then the escalationCode MAY be entered. This Event "catches" the escalation. If there is no escalationCode, then any escalation SHALL trigger the Event. If there is an escalationCode, then only an escalation that matches the escalationCode SHALL trigger the Event.
escalationRef: Escalation [0..1]	If the *Trigger* is an *Escalation*, then an *Escalation* payload MAY be provided

Link Event Definition

A Link Event is a mechanism for connecting two sections of a Process. Link Events can be used to create looping situations or to avoid long Sequence Flow lines. The use of Link Events is limited to a single Process level (i.e., they cannot link a *parent* Process with a Sub-Process).

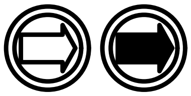


Figure 10-78 – Link Events

Paired Link Events can also be used as "Off-Page Connectors" for printing a Process across multiple pages. They can also be used as generic "Go To" objects within the Process level. There can be multiple *source* Link Events, but there can only be one *target* Link Event. When used to "catch" from the *source* Link, the Event marker will be unfilled (see the top figure on the right). When used to "throw" to the *target* Link, the Event marker will be filled (see the bottom figure on the right).

Since Process models often extend beyond the length of one printed page, there is often a concern about showing how Sequence Flow connections extend across the page breaks. One solution that is often employed is the use of Off-Page connectors to show where one page leaves off and the other begins. BPMN provides Intermediate Events of type Link for use as Off-Page connectors (see Figure 10-79--Note that the figure shows two different printed pages, not two Pools in one diagram). A pair of Link Events is used. One of the pair is shown at the end of one page. This Event is named and has an *incoming* Sequence Flow and no *outgoing* Sequence Flow. The second Link Event is at the beginning of the next page, shares the same name, and has an *outgoing* Sequence Flow and no *incoming* Sequence Flow.

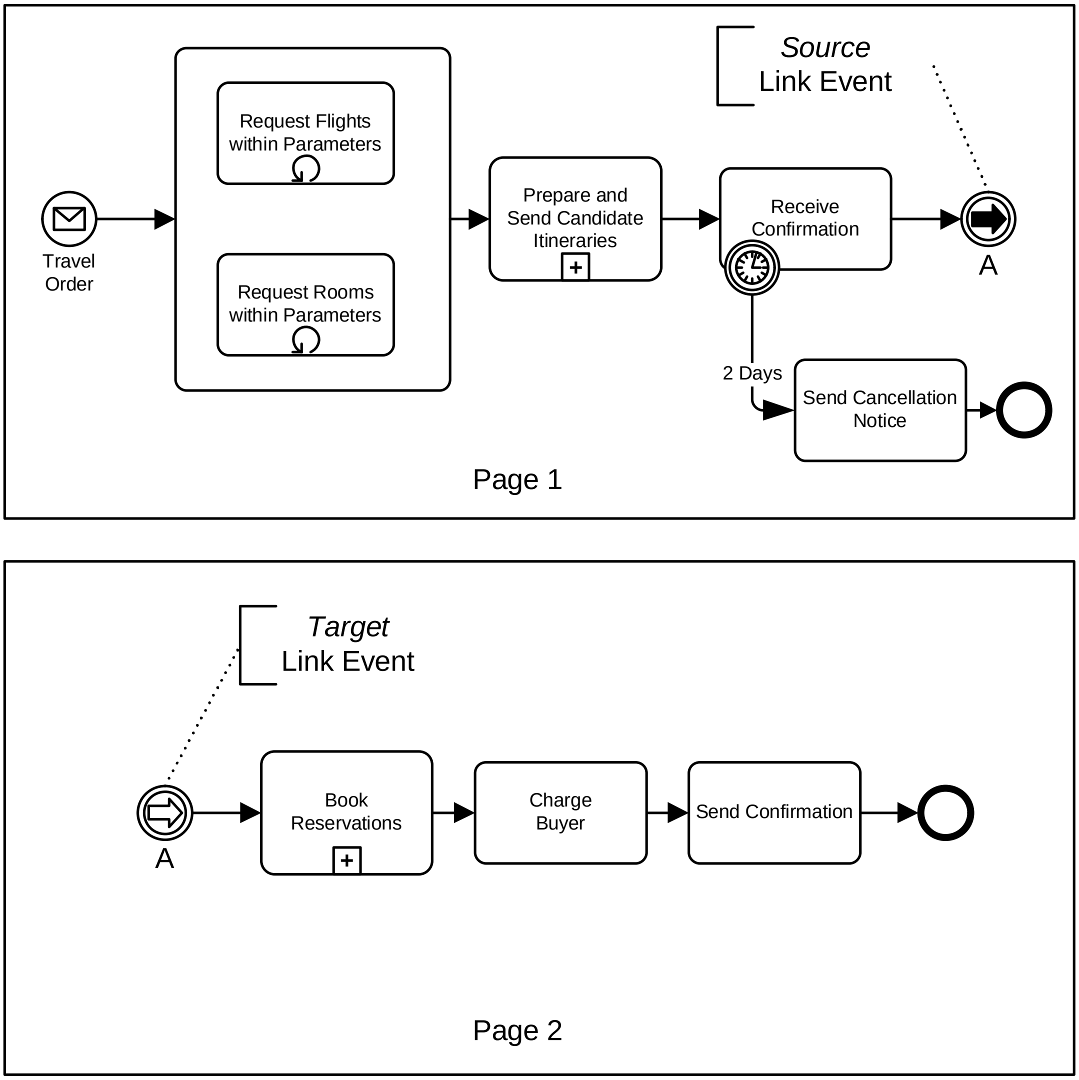


Figure 10-79 – Link Events Used as Off-Page Connector

Another way that Link Events can be used is as “Go To” objects. Functionally, they would work the same as for Off-Page Connectors (described above), except that they could be used anywhere in the diagram--on the same page or across multiple pages. The general idea is that they provide a mechanism for reducing the length of Sequence Flow lines. Some modelers may consider long lines as being hard to follow or trace. Go To Objects can be used to avoid very long Sequence Flow (see Figure 10-80 and Figure 10-81). Both diagrams will behave equivalently. For Figure 10-81, if the “Order Rejected” path is taken from the Decision, then the *token* traversing the Sequence Flow would reach the *source* Link Event and then “jump” to the *target* Link Event and continue down the Sequence Flow. The Process would continue as if the Sequence Flow had directly connected the two objects.

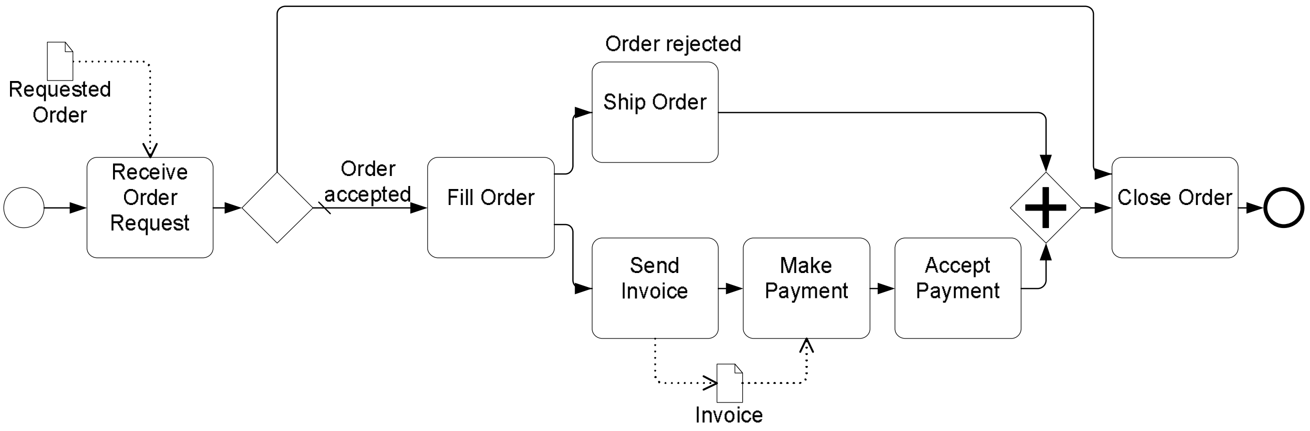


Figure 10-80 – Process with Long Sequence Flow

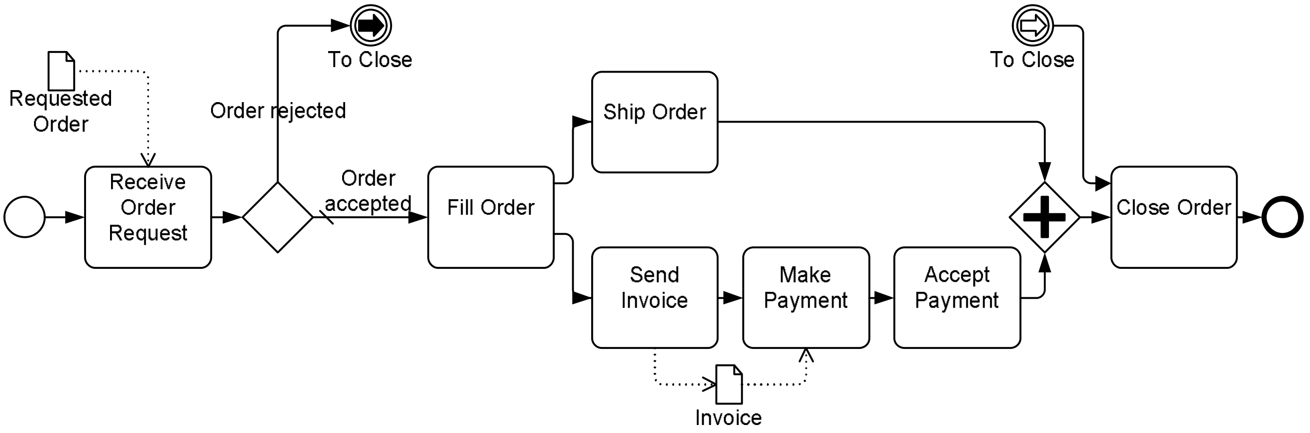


Figure 10-81 – Process with Link Intermediate Events Used as Go To Objects

Some methodologies prefer that all Sequence Flow only move in one direction; that is, forward in time. These methodologies do not allow Sequence Flow to connect directly to upstream objects. Some consistency in modeling can be gained by such a methodology, but situations that require looping become a challenge. Link Events can be used to make upstream connections and create *loops* without violating the Sequence Flow direction restriction (see Figure 10-82).

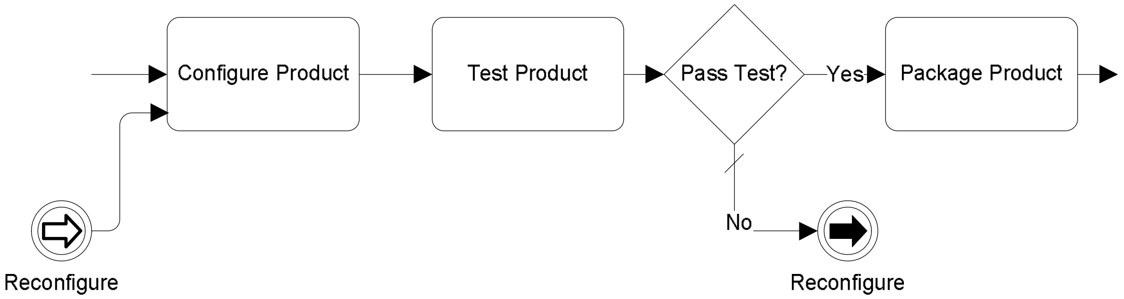


Figure 10-82 – Link Events Used for looping

The `LinkEventDefinition` element inherits the attributes and model associations of `BaseElement` (see Table 8-5) through its relationship to the `EventDefinition` element (see page 269). Table 10-91 presents the additional attributes of the `LinkEventDefinition` element:

Table 10-91 – LinkEventDefinition attributes

Attribute Name	Description/Usage
name: string	If the *Trigger* is a *Link*, then the name MUST be entered.

Message Event Definition

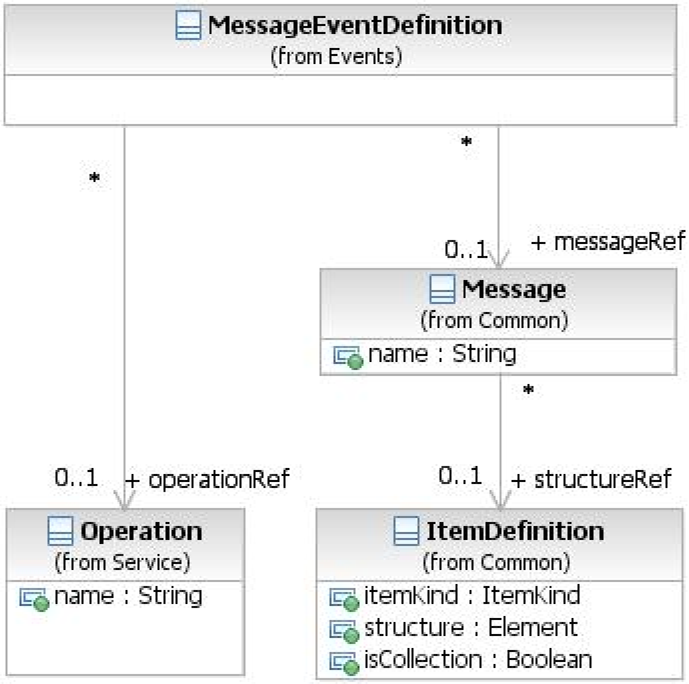


Figure 10-83 – MessageEventDefinition Class Diagram

Figure 10-84

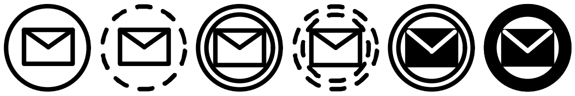


Figure 10-84 – Message Events

The `MessageEventDefinition` element inherits the attributes and model associations of `BaseElement` (see Table 8-5) through its relationship to the `EventDefinition` element (see page 269). Table 10-92 presents the additional model associations of the `MessageEventDefinition` element:

Table 10-92 – MessageEventDefinition model associations

Attribute Name	Description/Usage
messageRef: Message [0..1]	The Message MUST be supplied (if the processType attribute of the Process is set to executable).
operationRef: Operation [0..1]	This attribute specifies the operation that is used by the Message Event. It MUST be specified for executable Processes.

Multiple Event

For a Start Event:

If the *Trigger* is Multiple, there are multiple ways of starting the Process. Only one of them is necessary to trigger the start of the Process. The EventDefinition subclasses will define which *Triggers* apply

For an End Event:

If the *Result* is Multiple, there are multiple consequences of ending the Process. All of them will occur. The EventDefinition subclasses will define which *Results* apply

For an Intermediate Event within *normal flow*:

If the *Trigger* is Multiple, only one EventDefinition is required to *catch* the trigger. When used to *throw*, all of the EventDefinitions are considered and the subclasses will define which *Results* apply

For an Intermediate Event attached to the boundary of an Activity:

If the *Trigger* is Multiple, only one EventDefinition is required to "catch" the trigger.

Figure 10-85

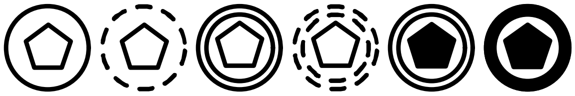


Figure 10-85 – Multiple Events

None Event

None Events are Events that do not have a defined EventDefinition. There are three (3) variations of None Events: a Start Event, a *catch* Intermediate Event, and an End Event (see Figure 10-86).

- The None Start Event MAY be used for a *top-level* Process or any type of Sub-Process (except an Event Sub-Process)

- The None Start Event MAY NOT be used for an Event Sub-Process.
- The *catch* None Intermediate Event MUST only be used in *Normal Flow* and, thus, MAY NOT be attached to the boundary of an Activity.
- The None End Event MAY be used within any Sub-Process or Process.

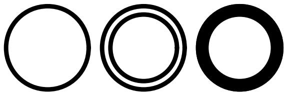


Figure 10-86 – None Events

Parallel Multiple Event

For a Start Event:

If the *trigger* is Multiple, there are multiple triggers required to start the Process. All of them are necessary to trigger the start of the Process. The EventDefinition subclasses will define which *triggers* apply. In addition, the parallelMultiple attribute of the Start Event MUST be set to *true*.

For an Intermediate Event within *normal flow*:

If the *trigger* is Multiple, all of the defined EventDefinitions are required to trigger the Event. In addition, the parallelMultiple attribute of the Intermediate Event MUST be set to *true*.

For an Intermediate Event attached to the boundary of an Activity:

If the *trigger* is Multiple, all of the defined EventDefinitions are required to trigger the Event. In addition, the parallelMultiple attribute of the Intermediate Event MUST be set to *true*.

Figure 10-85 shows the variations of Parallel Multiple Events.

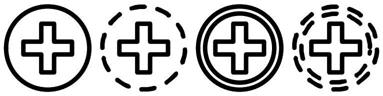


Figure 10-87 – Multiple Events

Signal Event

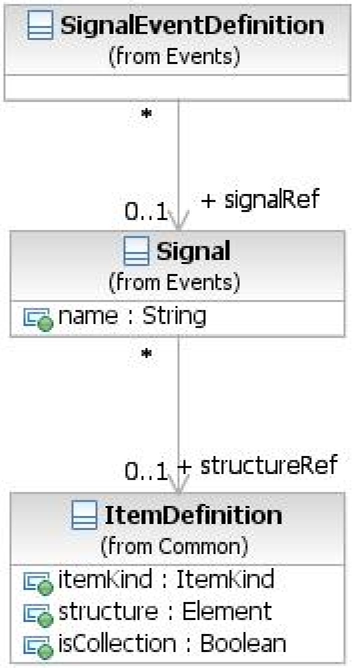


Figure 10-88 – SignalEventDefinition Class Diagram

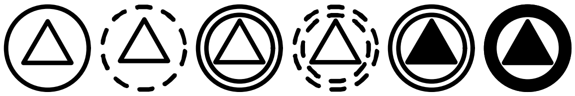


Figure 10-89 – Signal Events

The `SignalEventDefinition` element inherits the attributes and model associations of `BaseElement` (see Table 8-5) through its relationship to the `EventDefinition` element (see page 269). Table 10-93 presents the additional model associations of the `ConditionalSignalDefinition` element:

Table 10-93 – SignalEventDefinition model associations

Attribute Name	Description/Usage
signalRef: Signal	If the Trigger is a Signal, then a Signal is provided

Terminate Event

Figure 10-90

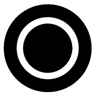


Figure 10-90 – Terminate Event

The CancelEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269).

Timer Event

Figure 10-91

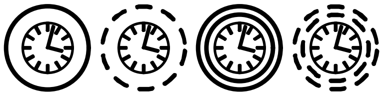


Figure 10-91 – Timer Events

The TimerEventDefinition element inherits the attributes and model associations of BaseElement (see Table 8-5) through its relationship to the EventDefinition element (see page 269). Table 10-94 presents the additional model associations of the TimerEventDefinition element:

Table 10-94 – TimerEventDefinition model associations

Attribute Name	Description/Usage
timeDate: Expression [0..1]	If the Trigger is a Timer, then a timeDate MAY be entered. If a timeDate is not entered, then a timeCycle MUST be entered (see attribute below—if the processType attribute of the Process is set to executable).
timeCycle: Expression [0..1]	If the Trigger is a Timer, then a timeCycle MAY be entered. If a timeCycle is not entered, then a timeDate MUST be entered (see attribute above—if the processType attribute of the Process is set to executable).

10.4.6. Handling Events

BPMN provides advanced constructs for dealing with Events that occur during the execution of a Process (i.e., the “catching” of an Event). Furthermore, BPMN supports the explicit creation of an Event in the Process (i.e., the “throwing” of an Event). Both *catching* and *throwing* of an Event as well as the resulting Process behavior is referred to as *Event handling.* There are three (3) types of *Event handlers*: those that start a Process, those that are part of the normal Sequence Flow, and those that are attached to Activities, either via boundary Events or via separate *inline handlers* in case of an Event Sub-Process.

Handling Start Events

There are multiple ways in which a Process can be started. For single Start Events, handling consists of starting a new Process *instance* each time the Event occurs. Sequence Flow leaving the Event is then followed as usual. . For multiple Start Events, BPMN supports several modeling scenarios that can be applied depending on the scenario.

Exclusive start: the most common scenario for starting a Process is its instantiation by exactly one out of many possible Start Events. Each occurrence of one of these Events will lead to the creation of a new Process *instance*. The following example shows two Events connected to a single Activity (see Figure 10-92). At runtime, each occurrence of one of the Events will lead to the creation of a new *instance* of the Process *instance* and activation of the Activity. Note that a single Multiple Start Event that contains the Message Event Definitions would behave in the same way.

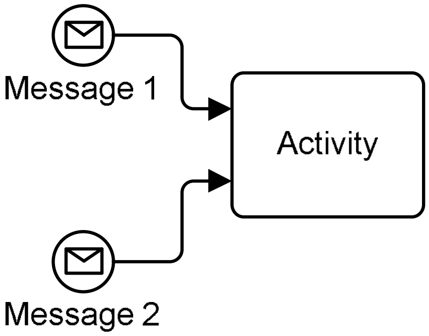


Figure 10-92 – Exclusive start of a Process

A Process can also be started via an Event-Based Gateway, as in the following example:

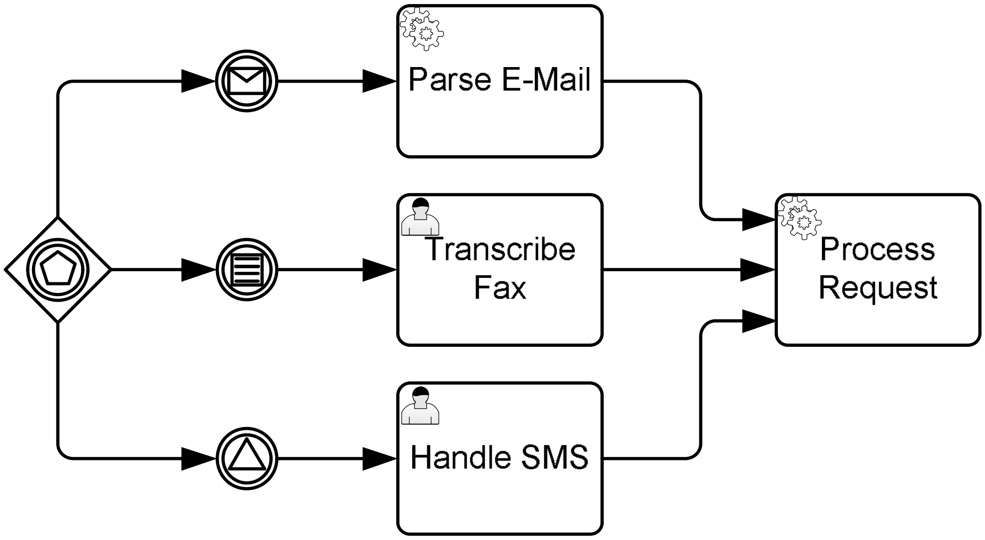


Figure 10-93 – A Process initiated by an Event-Based Gateway

In that case, the first matching Event will create a new *instance* of the Process, and waiting for the other Events originating from the same decision stops, following the usual semantics of the Event-Based

Exclusive Gateway. Note that this is the only scenario where a Gateway can exist without an *incoming* Sequence Flow.

It is possible to have multiple groups of Event-Based Gateways starting a Process, provided they participate in the same Conversation and hence share the same correlation information. In that case, one Event out of each group needs to arrive; the first one creates a new Process *instance*, while the subsequent ones are routed to the existing *instance*, which is identified through its correlation information.

Event synchronization: if the modeler requires several disjoint Start Events to be merged into a single Process *instance*, then the following notation must be applied:

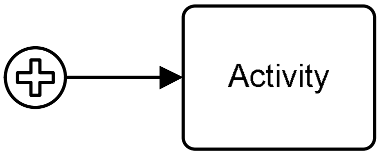


Figure 10-94 – Event synchronization at Process start

The Parallel Start Event may group several disjoint Start Events each of which must occur once in order for an *Instance* of the Process to be created. Sequence Flow leaving the Event is then followed as usual.

See page 442 for the execution semantics for the *Event Handling* of Start Events.

Handling Events within normal Sequence Flow (Intermediate Events)

For Intermediate Events, the handling consists of waiting for the Event to occur. Waiting starts when the Intermediate Event is reached. Once the Event occurs, it is consumed. Sequence flow leaving the Event is followed as usual.

Handling Events attached to an Activity (Intermediate boundary Events and Event Sub-Processes)

For boundary Events, handling consists of consuming the Event occurrence and either canceling the Activity the Event is attached to, followed by normal Sequence Flow leaving that Activity, or by running an *Event Handler* without canceling the Activity (only for Message, Signal, Timer and Conditional Events, not for Error Events).

An interrupting boundary Event is defined by a *true* value of its cancelActivity attribute. Whenever the Event occurs, the associated Activity is terminated. A downstream *token* is then generated, which activates the next element of the Process (connected to the Event by an unconditional Sequence Flow called an *Exception Flow*).

For non-interrupting boundary Events, the cancelActivity attribute is set to *false*. Whenever the Event occurs, the associated Activity continues to be active. As a *token* is generated for the Sequence Flow from the boundary Event in parallel to the continuing execution of the Activity, care must be taken when this flow is merged into the main flow of the Process – typically it should be ended with its own End Event.

The following example shows a fragment (see Figure 10-95) from a trip booking Process. It contains a Sub-Process that consists of a main part, and three Event Sub-Processes to deal with Events within the same context: an error Event Sub-Process that cancels the Sub-Process, a Message Event Sub-Process that updates the state of the Sub-Process while allowing it to continue, and a Compensation Event Sub-Process (see page 188).

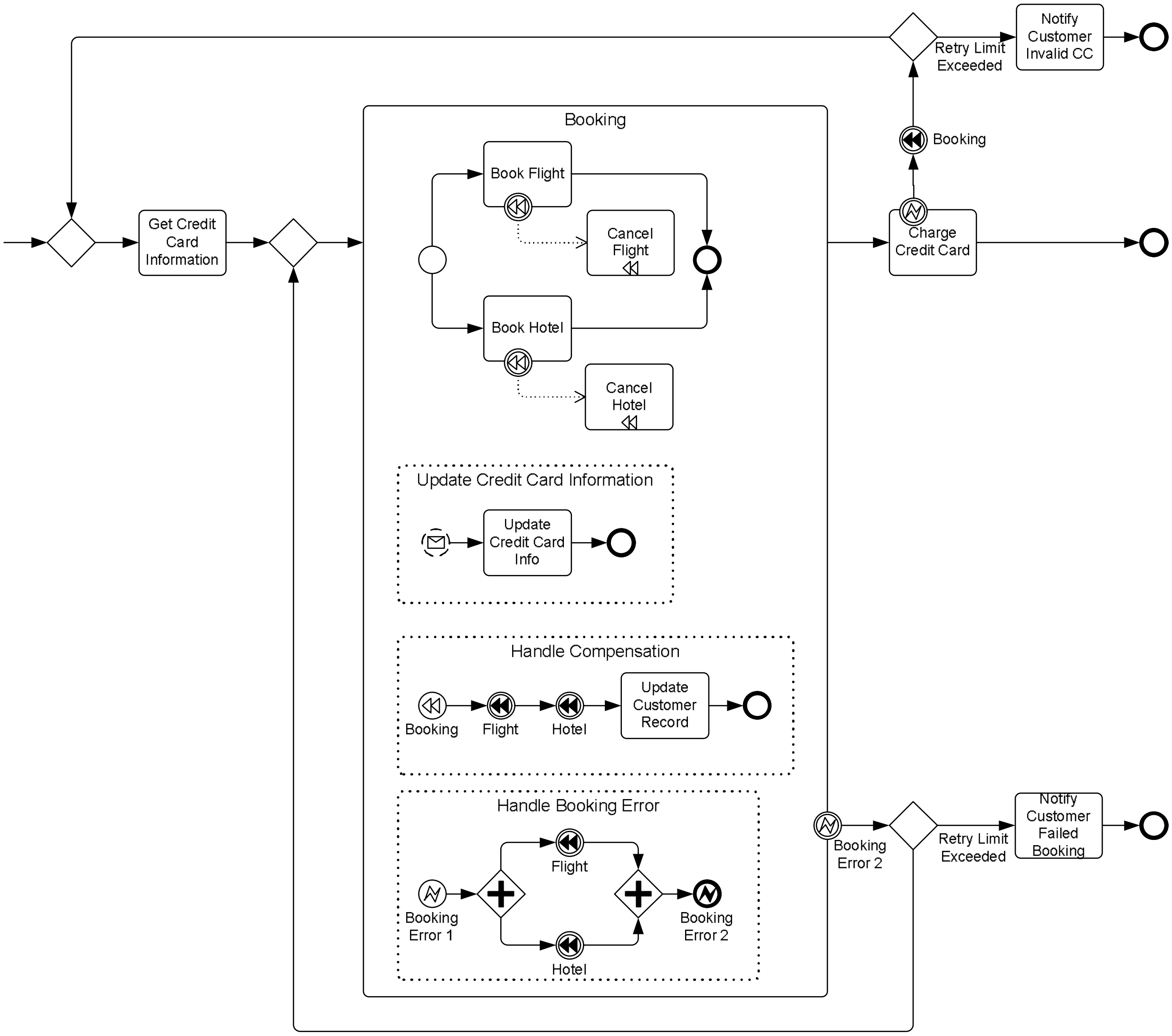


Figure 10-95 – Example of inline *Event Handling* via Event Sub-Processes

The following example (see Figure 10-96) shows the same fragment of that Process, using boundary Event handlers rather than inline Event Sub-Processes. Note that in this example, the handlers do not have access to the context of the "Booking" Sub-Process, as they run outside of it. Therefore, the actually *compensation* logic is shown as a black box.

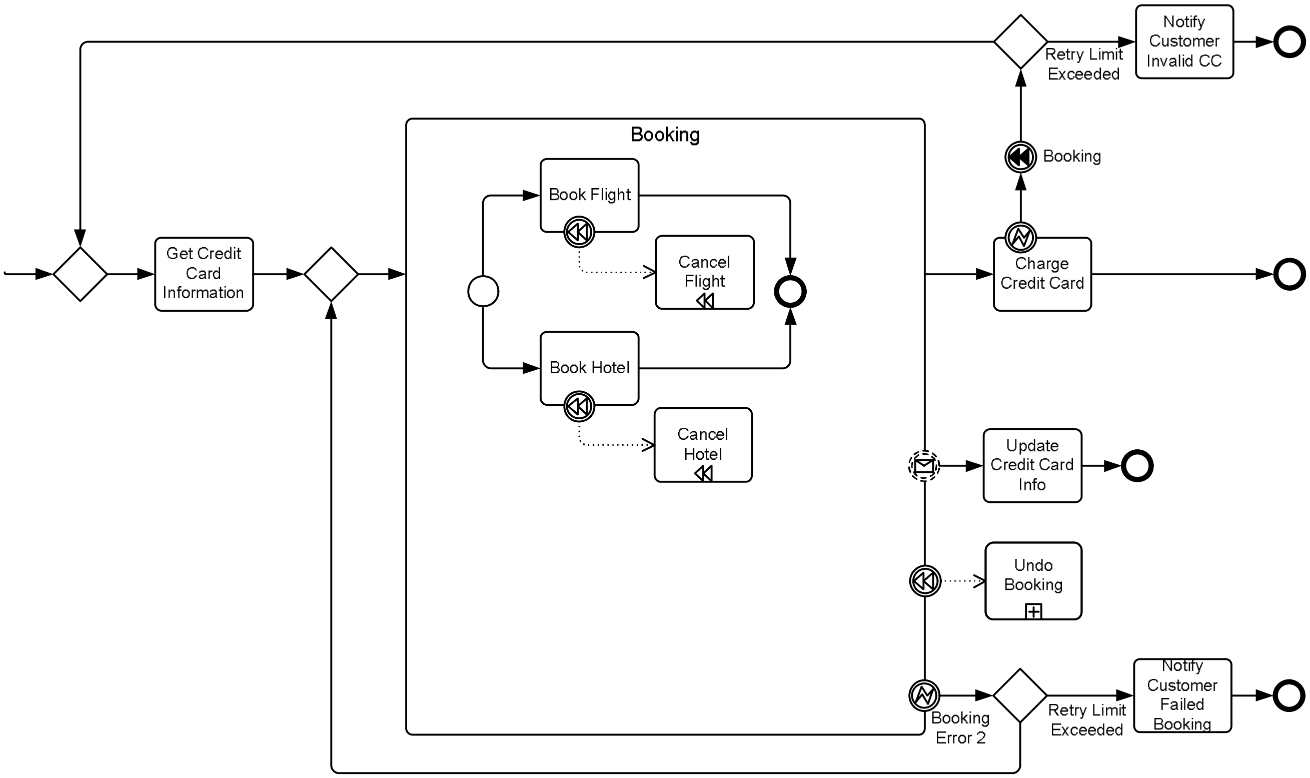


Figure 10-96 – Example of boundary *Event Handling*

Note that there is a distinction between *interrupting* and *non-interrupting* Events and the handling of these Events, which is described in the sections below. For an interrupting Event (Error, Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple), only one Event Sub-Process for the same Event Declaration may be modeled. This excludes any further non-interrupting handlers for that Event Declaration.

The reason for this restriction lies in the nature of interrupting Event Sub-Processes and boundary Events. They execute synchronously and after their completion, the hosting Activity is immediately terminated. This implies that only one such handler can be executed at a time. However, this does not restrict the modeler in specifying several interrupting handlers, if each handler refers to a different Event Declaration.

For non-interrupting Events (Escalation, Message, Signal, Timer, Conditional, Mulitple, and Parallel Multiple), an unlimited number of Event Sub-Processes for the same Event Declaration can be modeled and executed in parallel. At runtime, they will be invoked in a non-deterministic order. The same restrictions apply for boundary Events.

If for a given Sub-Process, both an inline Event Sub-Process and a boundary Event handler are modeled that Process the same EventDefinition, the following semantics apply:

- If the inline Event Sub-Process "rethrows" the Event after completion, the boundary Event is triggered
- If the inline Event Sub-Process completes without "rethrowing" the Event, the Activity is considered to have completed and normal Sequence Flow resumes. In other terms, the Event Sub-Process "absorbs" the Event.

Interrupting Event Handlers (Error, Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple)

Interrupting *Event Handlers* are those that have the cancelActivity attribute is set to *true*. Whenever the Event occurs, regardless of whether the Event is handled inline or on the boundary, the associated Activity is canceled. If an inline error handler is specified (in case of a Sub-Process), it is run within the context of that Sub-Process. If a boundary Error Event is present, Sequence Flow from that boundary Event is then followed.

In the example above, the "Booking" Sub-Process has an *Error handler* that defines what should happen in case a "Booking" Error occurs within the Sub-Process, namely, the already performed bookings are canceled using *compensation*. The *Error handler* is then continued outside the Sub-Process through a boundary Error Event.

Non-interrupting Event Handlers (Escalation, Message, Signal, Timer, Conditional, Multiple, and Parallel Multiple)

Interrupting *Event Handlers* are those that have the cancelActivity attribute is set to *false*.

For Event Sub-Processes, whenever the Event occurs it is consumed and the associated Event Sub-Process is performed. If there are several Events that happen in parallel, then they are handled concurrently, i.e., several Event Sub-Process *instances* are created concurrently. The *non-interrupting* Start Event indicates that the Event Sub-Process *instance* runs concurrently to the Sub-Process proper.

For boundary Events, whenever the Event occurs the handler runs concurrently to the Activity. If an Event Sub-Process is also specified for that Event (in case of a Sub-Process), it is run within the context of that Sub-Process. Then, Sequence Flow from the boundary Event is followed. As a *token* is generated for the Sequence Flow from the boundary Event in parallel to the continuing execution of the Activity, care must be taken when this flow is merged into the main flow of the Process – typically it should be ended with its own End Event.

In the example above, an *Event Handler* allows to update the credit card information during the "Booking" Sub-Process. It is triggered by a credit card information Message: such a Message can be received whenever the control flow is within the main body of the Sub-Process. Once such a Message is received, the Activities within the corresponding *Event Handler* run concurrently with the Activities within the body of the Sub-Process.

See page 442 for the exact semantics of boundary Intermediate Events and page 443 for the operational semantics of non-interrupting Event Sub-Processes.

Handling End Events

For a Terminate End Event, all remaining active Activities within the Process are terminated.

A Cancel End Event is only allowed in the context of a Transaction Sub-Process and, as such, cancels the Sub-Process and aborts an associated *Transaction* of the Sub-Process.

For all other End Events, the behavior associated with the EventDefinition is performed. When there are no further active Activities, then the Sub-Process or Process *instance* is completed. See page 443 for exact semantics.

10.4.7. Scopes

A *scope* describes the context in which execution of an Activity happens. This consists of:

- The set of Data Objects available (including DataInput and DataOutput)
- The set of Events available for *catching* or *throwing triggers*
- The set of Conversations going on in that *scope*

In general, a *scope* contains exactly one main flow of Activities which is started, when the *scope* gets activated. Vice versa, all Activities are enclosed by a *scope*. *Scopes* are hierarchically nested.

Scopes may have several *scope instances* at runtime. They are also hierarchically nested according to their generation. In a *scope instance* several *tokens* may be active.

Scope instances in turn have a **lifecycle**, containing amongst others the states:

- Activated
- In execution
- Completed
- In Compensation
- Compensation
- In Error
- In Cancellation
- Cancelled

BPMN has the following model elements with *scope* characteristics:

- Choreography
- Pool
- Sub-Process
- Task
- Activity
- Multi-instances body

Scopes are used to define the semantics of

- Visibility of Data Objects (including DataInput and DataOutput)
- Event resolution
- Starting/stopping of *token* execution

The Data Objects, Events and correlation keys described by a *scope* may be explicitly modeled or implicitly defined.

10.4.8. Events Package XML Schemas

Table 10-95 – BoundaryEvent XML schema

```
<xsd:element name="boundaryEvent" type="tBoundaryEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tBoundaryEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent">
            <xsd:attribute name="cancelActivity" type="xsd:boolean" default="true"/>
            <xsd:attribute name="attachedToRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-96 – CancelEventDefinition XML schema

```
<xsd:element name="cancelEventDefinition" type="tCancelEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCancelEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-97 – CatchEvent XML schema

```
<xsd:element name="catchEvent" type="tCatchEvent"/>
<xsd:complexType name="tCatchEvent" abstract="true">
      <xsd:complexContent>
        <xsd:extension base="tEvent">
            <xsd:sequence>
                <xsd:element ref="dataOutput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataOutputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="outputSet" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="eventDefinition" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="eventDefinitionRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="parallelMultiple" type="xsd:boolean" default="false"/>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-98 – CancelEventDefinition XML schema

```
<xsd:element name="cancelEventDefinition" type="tCancelEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCancelEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-99 – CompensateEventDefinition XML schema

```
<xsd:element name="compensateEventDefinition" type="tCompensateEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tCompensateEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="waitForCompletion" type="xsd:boolean"/>
            <xsd:attribute name="activityRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-100 – ConditionalEventDefinition XML schema

```
<xsd:element name="conditionalEventDefinition" type="tConditionalEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tConditionalEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:sequence>
                <xsd:element name="condition" type="tExpression"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-101 – ErrorEventDefinition XML schema

```
<xsd:element name="errorEventDefinition" type="tErrorEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tErrorEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="errorCode" type="xsd:string"/>
            <xsd:attribute name="errorRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-102 – Escalation XML schema

```
<xsd:element name="escalation" type="tEscalation" substitutionGroup="reusableElement"/>
<xsd:complexType name="tEscalation">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="structureRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-103 – EscalationEventDefinition XML schema

```
<xsd:element name="escalationEventDefinition" type="tEscalationEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tEscalationEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="escalationCode" type="xsd:string"/>
            <xsd:attribute name="escalationRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-104 – Event XML schema

```
<xsd:element name="event" type="tEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tEvent" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tFlowNode"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-105 – EventDefinition XML schema

```
<xsd:element name="eventDefinition" type="tEventDefinition"/>
<xsd:complexType name="tEventDefinition" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-106 – IntermediateCatchEvent XML schema

```
<xsd:element name="intermediateCatchEvent" type="tIntermediateCatchEvent"
        substitutionGroup="flowElement"/>
<xsd:complexType name="tIntermediateCatchEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-107 – IntermediateThrowEvent XML schema

```
<xsd:element name="intermediateThrowEvent" type="tIntermediateThrowEvent"
        substitutionGroup="flowElement"/>
<xsd:complexType name="tIntermediateThrowEvent">
    <xsd:complexContent>
        <xsd:extension base="tThrowEvent"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-108 – LinkEventDefinition XML schema

```
<xsd:element name="linkEventDefinition" type="tLinkEventDefinition" substitutionGroup="eventDefinition"/>
<xsd:complexType name="tLinkEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-109 – MessageEventDefinition XML schema

```
<xsd:element name="messageEventDefinition" type="tMessageEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tMessageEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:sequence>
                <xsd:element name="operationRef" type="xsd:QName" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
        <xsd:attribute name="messageRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-110 – Signal XML schema

```
<xsd:element name="signal" type="tSignal" substitutionGroup="reusableElement"/>
<xsd:complexType name="tSignal">
    <xsd:complexContent>
        <xsd:extension base="tRootElement">
            <xsd:attribute name="name" type="xsd:string"/>
            <xsd:attribute name="structureRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-111 – SignalEventDefinition XML schema

```
<xsd:element name="signalEventDefinition" type="tSignalEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tSignalEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:attribute name="signalRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-112 – StartEvent XML schema

```
<xsd:element name="startEvent" type="tStartEvent" substitutionGroup="flowElement"/>
<xsd:complexType name="tStartEvent">
    <xsd:complexContent>
        <xsd:extension base="tCatchEvent"/>
            </xsd:complexContent>
</xsd:complexType>
```

Table 10-113 – TerminateEventDefinition XML schema

```
<xsd:element name="terminateEventDefinition" type="tTerminateEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tTerminateEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-114 – ThrowEvent XML schema

```
<xsd:element name="throwEvent" type="tThrowEvent"/>
<xsd:complexType name="tThrowEvent" abstract="true">
      <xsd:complexContent>
        <xsd:extension base="tEvent">
            <xsd:sequence>
                <xsd:element ref="dataInput" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="dataInputAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="inputSet" minOccurs="0" maxOccurs="1"/>
                <xsd:element ref="eventDefinition" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="eventDefinitionRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Table 10-115 – TimerEventDefinition XML schema

```
<xsd:element name="timerEventDefinition" type="tTimerEventDefinition"
        substitutionGroup="eventDefinition"/>
<xsd:complexType name="tTimerEventDefinition">
    <xsd:complexContent>
        <xsd:extension base="tEventDefinition">
            <xsd:choice>
                <xsd:element name="timeDate" type="tExpression" minOccurs="0" maxOccurs="1"/>
                <xsd:element name="timeCycle" type="tExpression" minOccurs="0" maxOccurs="1"/>
            </xsd:choice>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

10.5. Gateways

Gateways are used to control how Sequence Flows interact as they converge and diverge within a Process. If the flow does not need to be controlled, then a Gateway is not needed. The term “Gateway” implies that there is a gating mechanism that either allows or disallows passage through the Gateway--that is, as *Tokens* arrive at a Gateway, they can be merged together on input and/or split apart on output as the Gateway mechanisms are invoked.

A Gateway is a diamond, which has been used in many flow chart notations for exclusive branching and is familiar to most modelers.

- A Gateway is a diamond that MUST be drawn with a single thin line (see Figure 10-97).
    - The use of text, color, size, and lines for a Gateway MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63 with the exception that:

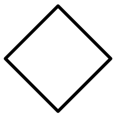


Figure 10-97 – A Gateway

Gateways, like Activities, are capable of consuming or generating additional *tokens*, effectively controlling the execution semantics of a given Process. The main difference is that Gateways do not represent ‘work’ being done and they are considered to have zero effect on the operational measures of the Process being executed (cost, time, etc.).

Gateways can define all the types of Business Process Sequence Flow behavior: Decisions/branching (exclusive, inclusive, and complex), merging, forking, and joining. Thus, while the diamond has been used traditionally for exclusive decisions, BPMN extends the behavior of the diamonds to reflect any type of Sequence Flow control. Each type of Gateway will have an internal indicator or marker to show the type of Gateway that is being used (see Figure 10-98).

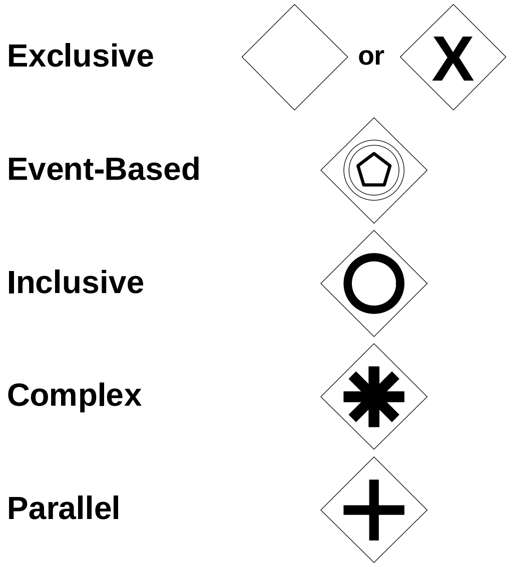


Figure 10-98 – The Different types of Gateways

The Gateway controls the flow of both diverging and converging Sequence Flow. That is, a single Gateway could have multiple input and multiple output flows. Modelers and modeling tools may want to enforce a best practice of a Gateway only performing one of these functions. Thus, it would take two sequential Gateways to first converge and then to diverge the Sequence Flow.

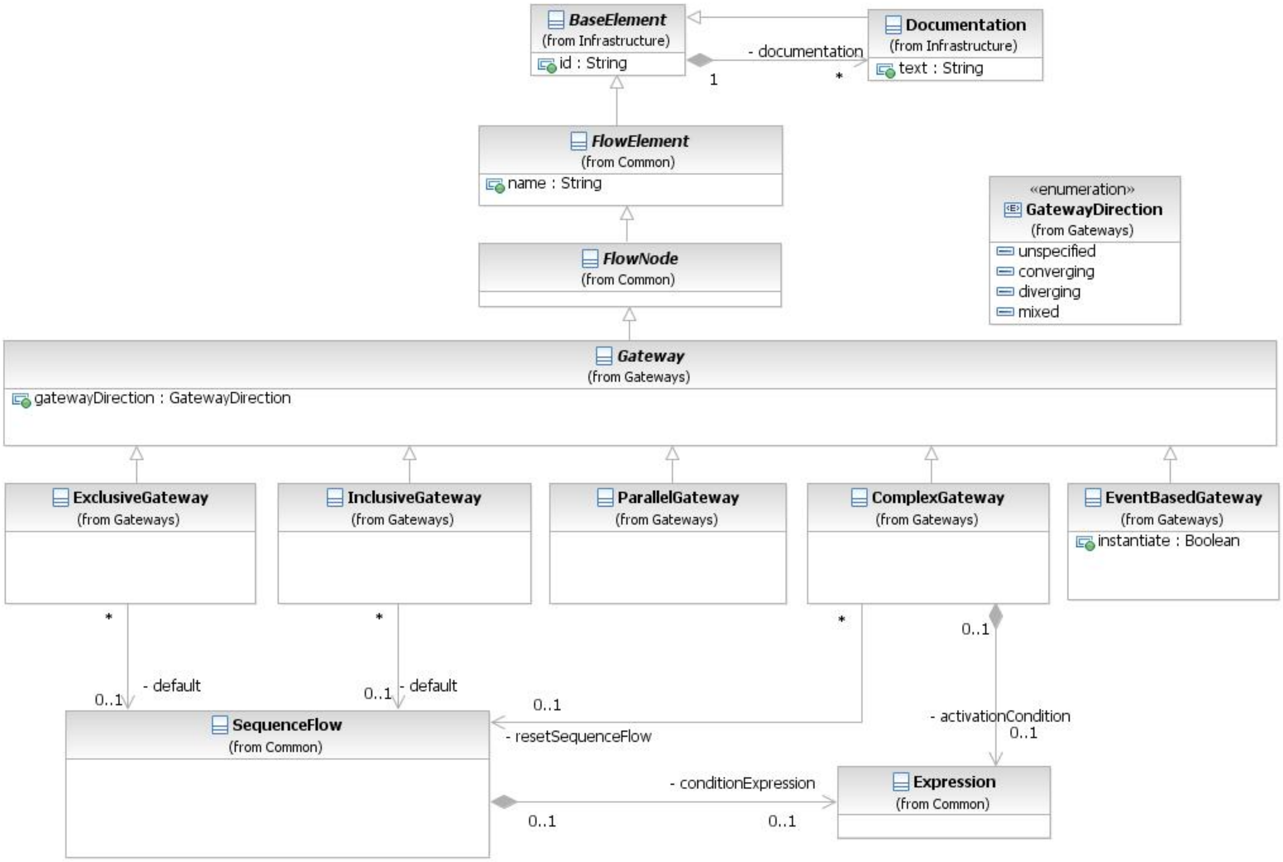


Figure 10-99 – Gateway class diagram

Gateways are described in this section on an abstract level. The execution semantics of Gateways is detailed on page 436.

10.5.1. Sequence Flow Considerations

Note – Although the shape of a Gateway is a diamond, it is not a requirement that *incoming* and *outgoing* Sequence Flow must connect to the corners of the diamond. Sequence Flow can connect to any position on the boundary of the Gateway shape.

This section applies to all Gateways. Additional Sequence Flow Connection rules may be specified for each type of Gateway in the sections below.

- A Gateway MAY be a target for Sequence Flow. It can have zero (0), one (1), or more *incoming* Sequence Flow.
  - If the Gateway does not have an *incoming* Sequence Flow, and there is no Start Event for the Process, then the Gateway's divergence behavior, depending on the type of Gateway (see below), SHALL be performed when the Process is instantiated.
- A Gateway MAY be a source of Sequence Flow; it can have zero (0), one (1), or more *outgoing* Sequence Flow.

- A Gateway MUST have either multiple *incoming* Sequence Flow or multiple *outgoing* Sequence Flow (i.e., it must merge or split the flow).
    - A Gateway with a `gatewayDirection` of `unspecified` MAY have both multiple *incoming* and *outgoing* Sequence Flow.
    - A Gateway with a `gatewayDirection` of `mixed` MUST have both multiple *incoming* and *outgoing* Sequence Flow.
    - A Gateway with a `gatewayDirection` of `converging` MUST have multiple *incoming* Sequence Flow, but MUST NOT have multiple *outgoing* Sequence Flow.
    - A Gateway with a `gatewayDirection` of `diverging` MUST have multiple *outgoing* Sequence Flow, but MUST NOT have multiple *incoming* Sequence Flow.

10.5.2. Exclusive Gateway

A diverging Exclusive Gateway (Decision) is used to create alternative paths within a Process flow. This is basically the "diversion point in the road" for a Process. For a given *instance* of the Process, only one of the paths can be taken.

A Decision can be thought of as a question that is asked at a particular point in the Process. The question has a defined set of alternative answers. Each question is associated with a condition expression that is associated with a Gateway's *outgoing* Sequence Flow.

- The Exclusive Gateway MAY use a marker that is shaped like an "X" and is placed within the Gateway diamond (see Figure 10-101) to distinguish it from other Gateways. This marker is not required (see Figure 10-100).
    - A diagram SHOULD be consistent in the use of the "X" internal indicator. That is, a diagram SHOULD NOT have some Gateways with an indicator and other Gateways without an indicator.

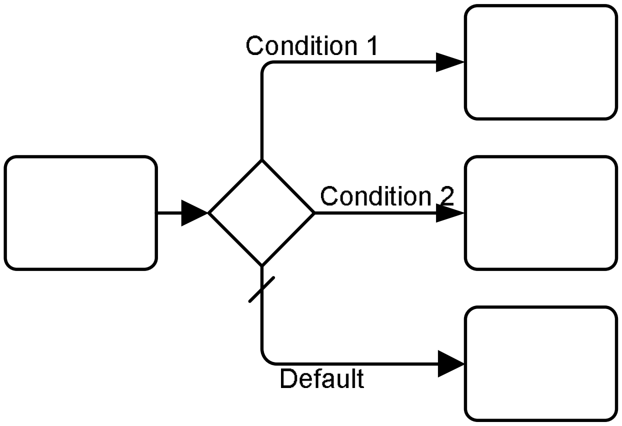


Figure 10-100 – An Exclusive Data-Based Decision (Gateway) Example without the Internal Indicator

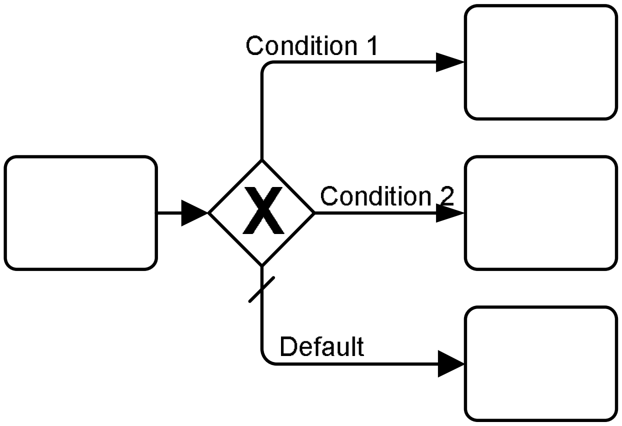


Figure 10-101 – A Data-Based Exclusive Decision (Gateway) Example with the Internal Indicator

Note – as a modeling preference, the Exclusive Gateways shown in examples within this specification will be shown without the internal indicator.

A default path can optionally be identified, to be taken in the event that none of the conditional expressions evaluate to *true*. If a default path is not specified and the Process is executed such that none of the conditional expressions evaluates to *true*, a runtime exception occurs.

A converging Exclusive Gateway is used to merge alternative paths. Each incoming control flow *token* is routed to the *outgoing* Sequence Flow without synchronization.

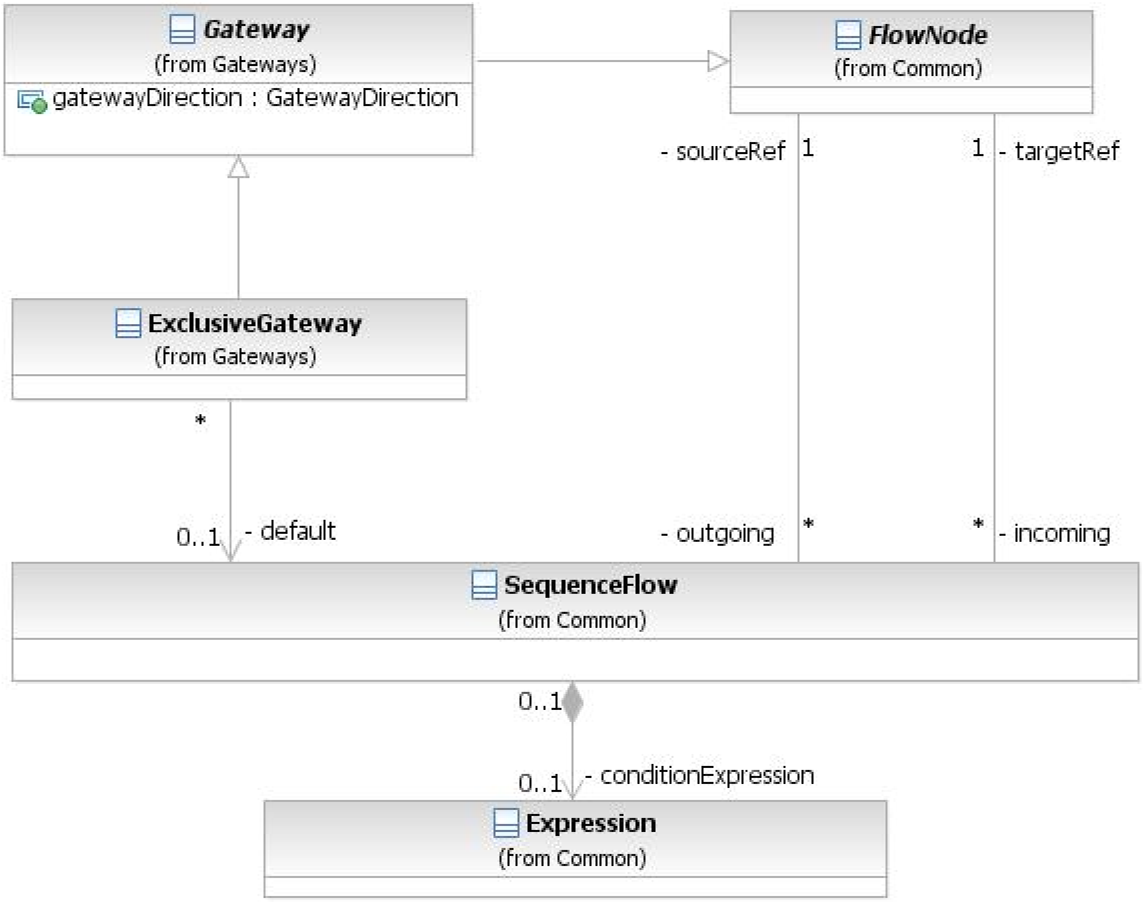


Figure 10-102 – Exclusive Gateway class diagram

The Exclusive Gateway element inherits the attributes and model associations of Gateway (see Table 8-47). Table 10-116 presents the additional attributes and model associations of the Exclusive Gateway element:

Table 10-116 – ExclusiveGateway Attributes & Model Associations

Attribute Name	Description/Usage
default: SequenceFlow [0..1]	The Sequence Flow that will receive a *Token* when none of the conditionExpressions on other *outgoing* Sequence Flow evaluate to *true*. The *default* Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.

10.5.3. Inclusive Gateway

A diverging Inclusive Gateway (Inclusive Decision) can be used to create alternative but also parallel paths within a Process flow. Unlike the Exclusive Gateway, all condition expressions are evaluated. The *true* evaluation of one condition expression does not exclude the evaluation of other condition expressions. All Sequence Flow with a *true* evaluation will be traversed by a *Token*. Since each path is considered to be independent, all combinations of the paths may be taken, from zero to all. However, it should be designed so that at least one path is taken.

- The Inclusive Gateway MUST use a marker that is in the shape of a circle or an "O" and is placed within the Gateway diamond (see Figure 10-103) to distinguish it from other Gateways

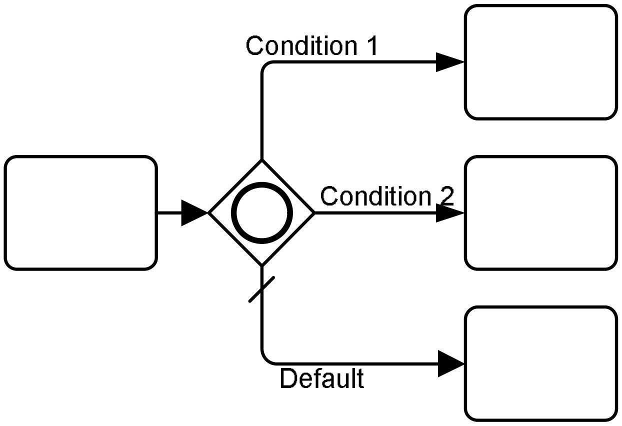


Figure 10-103 – An example using an Inclusive Gateway

A default path can optionally be identified, to be taken in the event that none of the conditional expressions evaluate to *true*. If a default path is not specified and the Process is executed such that none of the conditional expressions evaluates to *true*, a runtime exception occurs.

A converging Inclusive Gateway is used to merge a combination of alternative and parallel paths. A control flow *Token* arriving at an Inclusive Gateway may be synchronized with some other *Tokens* that arrive later at this Gateway. The precise synchronization behavior of the Inclusive Gateway can be found on page 438.

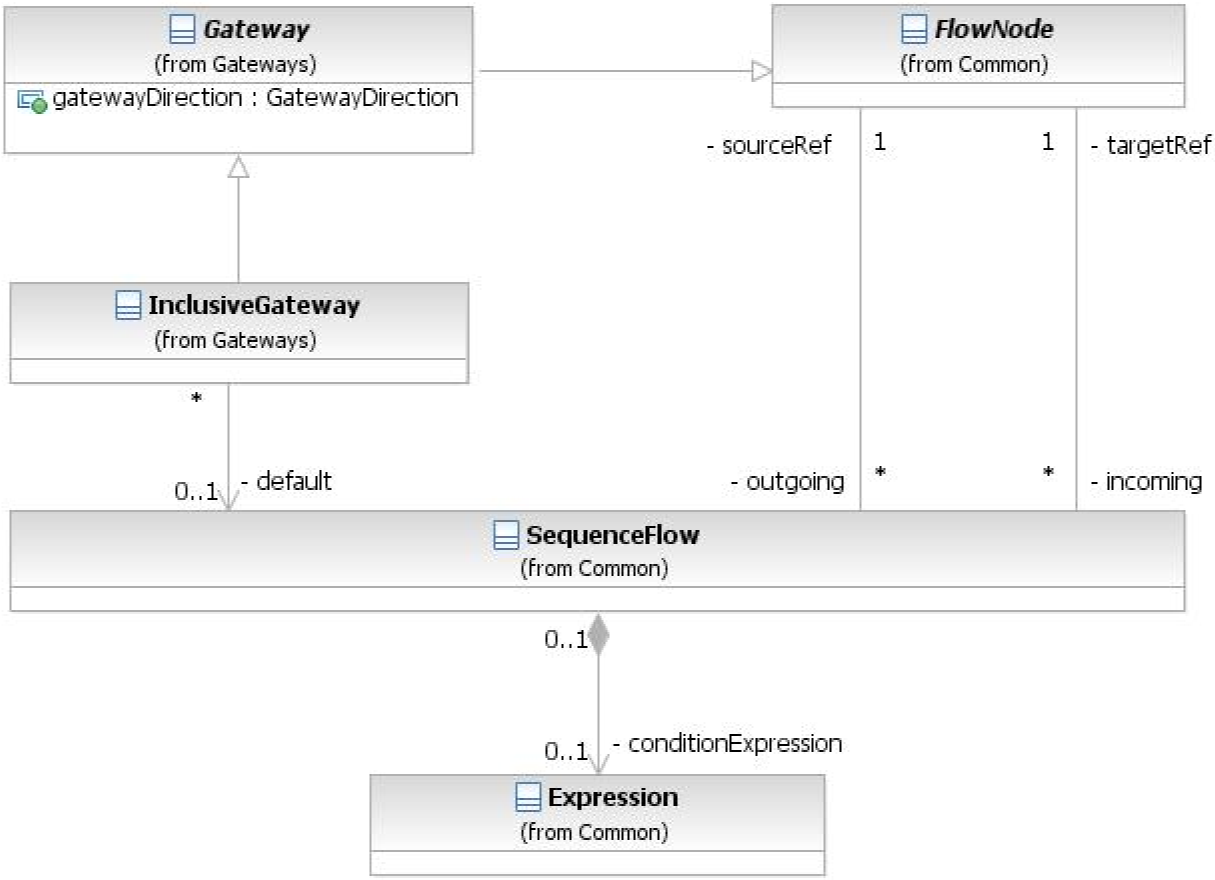


Figure 10-104 – Inclusive Gateway class diagram

The Inclusive Gateway element inherits the attributes and model associations of Gateway (see Table 8-47). Table 10-117 presents the additional attributes and model associations of the Inclusive Gateway element:

Table 10-117 – InclusiveGateway Attributes & Model Associations

Attribute Name	**Description/Usage**
default: SequenceFlow [0..1]	The Sequence Flow that will receive a *Token* when none of the conditionExpressions on other Sequence Flow evaluate to *true*. The *default* Sequence Flow should not have a conditionExpression. Any such `Expression` SHALL be ignored.

10.5.4. Parallel Gateway

A Parallel Gateway is used to synchronize (combine) parallel flows and to create parallel flows.

- The Parallel Gateway MUST use a marker that is in the shape of a plus sign and is placed within the Gateway diamond (see Figure 10-105) to distinguish it from other Gateways

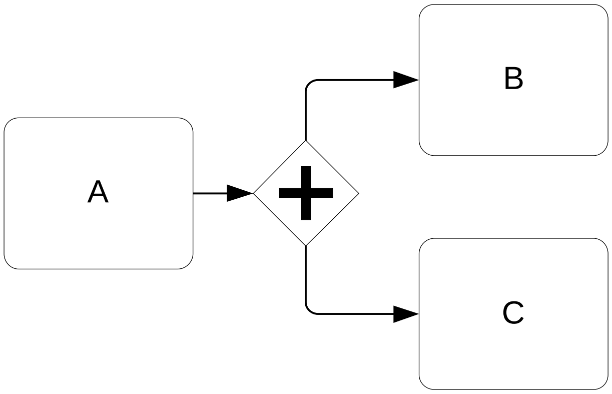


Figure 10-105 – An example using an Parallel Gateway

Parallel Gateways are used for synchronizing parallel flow (see Figure 10-106).

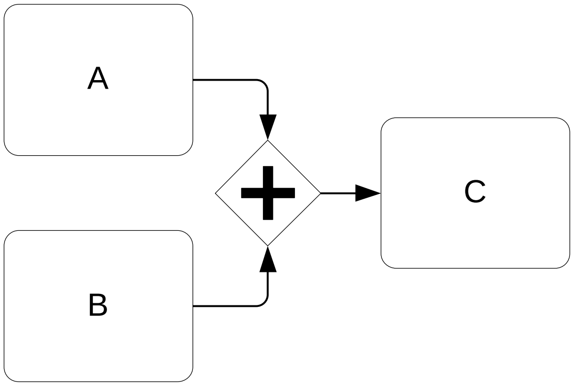


Figure 10-106 – An example of a synchronizing Parallel Gateway

A Parallel Gateway creates parallel paths without checking any conditions; each *outgoing* Sequence Flow receives a *token* upon execution of this Gateway. For incoming flows, the Parallel Gateway will wait for all incoming flows before triggering the flow through its *outgoing* Sequence Flows.

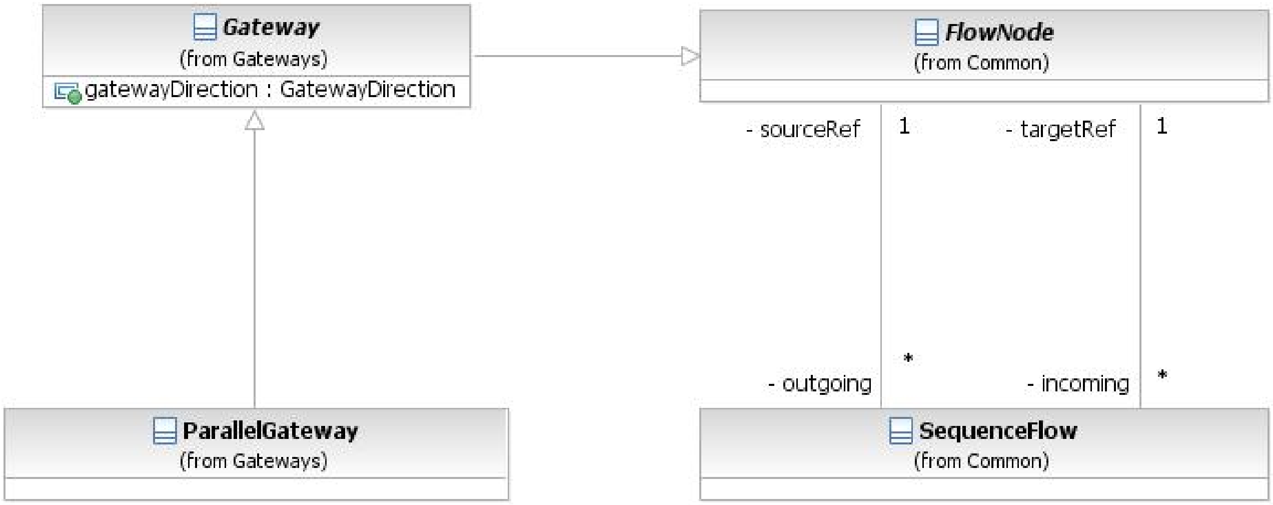


Figure 10-107 – Parallel Gateway class diagram

The Parallel Gateway element inherits the attributes and model associations of Gateway (see Table 8-47), but adds no additional attributes or model associations.

10.5.5. Complex Gateway

The Complex Gateway can be used to model complex synchronization behavior. An Expression activationCondition is used to describe the precise behavior. For example, this Expression could specify that *tokens* on three out of five *incoming* Sequence Flow are needed to activate the Gateway. What *tokens* are produced by the Gateway is determined by conditions on the *outgoing* Sequence Flow as in the split behavior of the Inclusive Gateway. If *token* arrive later on the two remaining Sequence Flow, those *tokens* cause a reset of the Gateway and new *token* can be produced on the *outgoing* Sequence Flow. To determine whether it needs to wait for additional *tokens* before it can reset, the Gateway uses the synchronization semantics of the Inclusive Gateway.

- The Complex Gateway MUST use a marker that is in the shape of an asterisk and is placed within the Gateway diamond (see Figure 10-108) to distinguish it from other Gateways

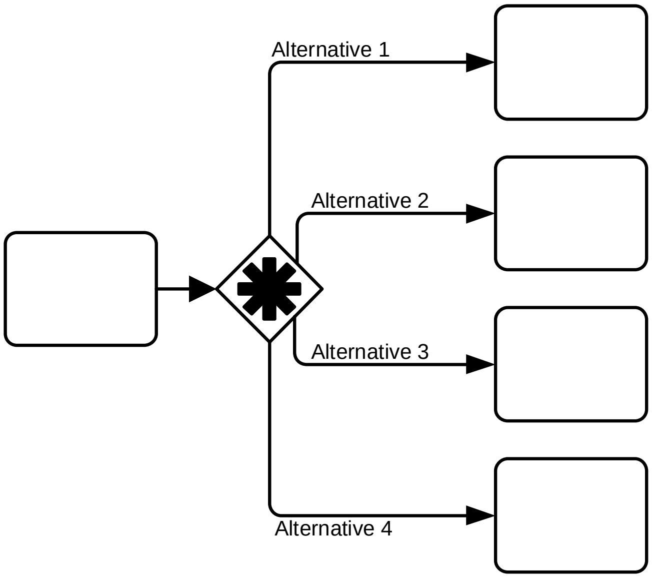


Figure 10-108 – An example using a Complex Gateway

The Complex Gateway has, in contrast to other Gateways, an internal state, which is represented by the Boolean instance attribute waitingForStart, which is initially *true* and becomes *false* after activation. This attribute can be used in the conditions of the *outgoing* Sequence Flow to specify where *tokens* are produced upon activation and where *tokens* are produced upon reset. It is recommended that each *outgoing* Sequence Flow may either get a *token* upon activation or upon reset but not both. At least one *outgoing* Sequence Flow should receive a *token* upon activation but no *token* may be produced upon reset.

Figure 10-109 shows the class diagram for the Complex Gateway.

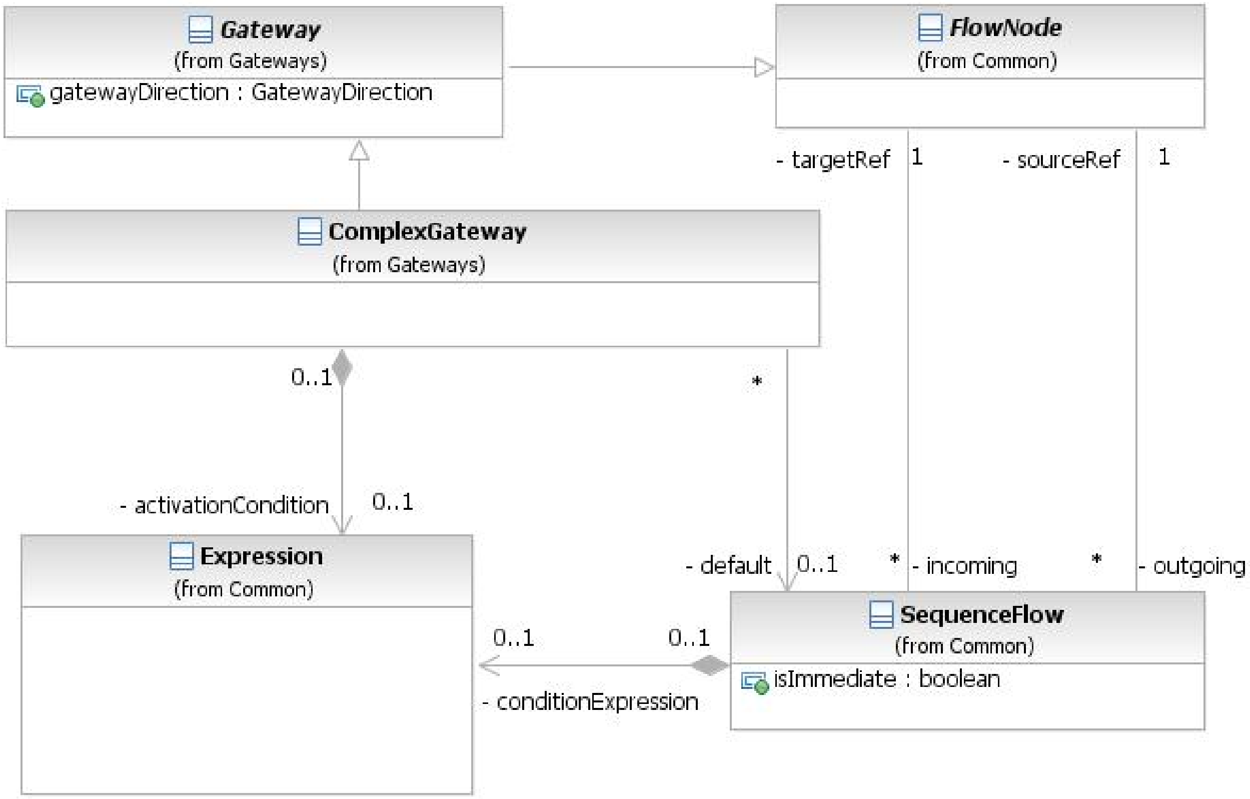


Figure 10-109 – Complex Gateway class diagram

The Complex Gateway element inherits the attributes and model associations of Gateway (see Table 8-47). Table 10-118 presents the additional model associations of the Complex Gateway element:

Table 10-118 – Complex Gateway model associations

Attribute Name	Description/Usage
activationCondition: Expression [0..1]	Determines which combination of incoming *tokens* will be synchronized for activation of the Gateway.
default: SequenceFlow [0..1]	The Sequence Flow that will receive a *token* when none of the conditionExpressions on other Sequence Flow evaluate to *true*. The *default* Sequence Flow should not have a conditionExpression. Any such Expression SHALL be ignored.

Table 10-119 – Instance Attributes related to the Complex Gateway

Attribute Name	Description/Usage
activationCount: integer	Refers at runtime to the number of *tokens* that are present on an *incoming* Sequence Flow of the Complex Gateway.
waitingForStart: boolean = True	Represents the internal state of the Complex Gateway. It is either waiting for start (=*true*) or waiting for reset (=*false*).

10.5.6. Event-Based Gateway

The Event-Based Gateway represents a branching point in the Process where the alternative paths that follow the Gateway are based on Events that occur, rather than the evaluation of Expressions using Process data (as with an Exclusive or Inclusive Gateway). A specific Event, usually the receipt of a Message, determines the path that will be taken. Basically, the *decision* is made by another *Participant*, based on data that is not visible to Process, thus, requiring the use of the Event-Based Gateway.

For example, if a company is waiting for a response from a customer they will perform one set of Activities if the customer responds "Yes" and another set of Activities if the customer responds "No." The customer's response determines which path is taken. The identity of the Message determines which path is taken. That is, the "Yes" Message and the "No" Message are different Messages—i.e., they are not the same Message with different values within a property of the Message. The receipt of the Message can be modeled with an Intermediate Event with a Message *Trigger* or a Receive Task. In addition to Messages, other *Triggers* for Intermediate Events can be used, such as Timers.

The Event Gateway shares the same basic shape of the Gateways, a diamond, with a marker placed within the diamond to indicate variations of the Gateway.

- An Event Gateway is a diamond that MUST be drawn with a single thin line.
  - The use of text, color, size, and lines for a Start Event MUST follow the rules defined in Section "Use of Text, Color, Size, and Lines in a Diagram" on page 63.
- The marker for the Event Gateway MUST look like a *catch* Multiple Intermediate Event (see Figure 10-110).

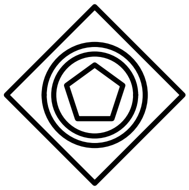


Figure 10-110 – Event-Based Gateway

Unlike other Gateways, the behavior of the Event Gateway is determined by a configuration of elements, rather than the single Gateway.

- An Event Gateway MUST have two (2) or more *outgoing* Sequence Flow.

- The *outgoing* Sequence Flow of the Event Gateway MUST NOT have a conditionExpression.

The objects that are on the target end of the Gateway's *outgoing* Sequence Flow are part of the configuration of the Gateway.

- Event-Based Gateways are configured by having *outgoing* Sequence Flow target an Intermediate Event or a Receive Task in any combination (see figures Figure 10-111 and Figure 10-112) except that:
  - If Message Intermediate Events are used in the configuration, then Receive Tasks MUST NOT be used in that configuration and vice versa.
    - Receive Tasks used in an Event Gateway configuration MUST NOT have any attached Intermediate Events.
  - Only the following Intermediate Event *triggers* are valid: Message, Signal, Timer, Conditional, and Multiple (which can only include the previous *triggers*). Thus, the follow Intermediate Event *triggers* are not valid: Error, Cancel, Compensation, and Link.
- Target elements in an Event Gateway configuration MUST NOT have any addition *incoming* Sequence Flow (other than that from the Event Gateway).

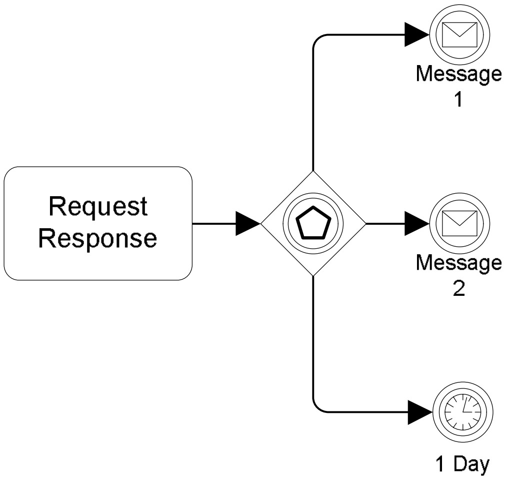


Figure 10-111 – An Event-Based Gateway example using Message Intermediate Events

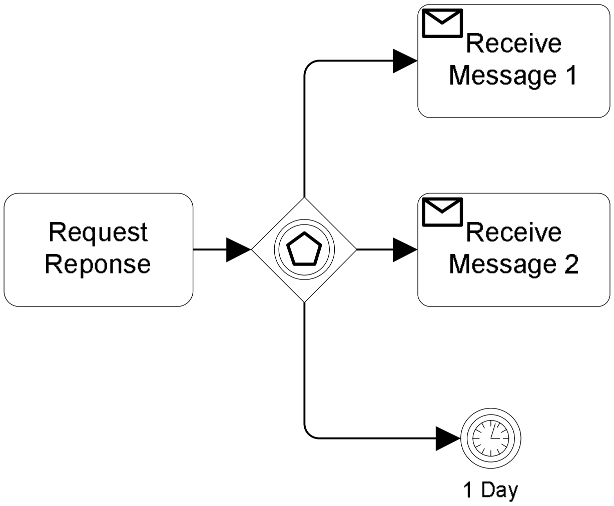


Figure 10-112 – An Event-Based Gateway example using Receive Tasks

When the first Event in the Event Gateway configuration is triggered, then the path that follows that Event will used (a *token* will be sent down the Event's outgoing Sequence Flow). All the remaining paths of the Event Gateway configuration will no longer be valid. Basically, the Event Gateway configuration is a race condition where the first Event that is triggered wins.

There are variations of the Event Gateway that can be used at the start of the Process. The behavior and marker of the Gateway will change.

Event Gateways can be used to instantiate a Process. By default the Gateway's instantiate attribute is *false*, but if set to *true*, then the Process is instantiated when the first Event of the Gateway's configuration is triggered.

- If the Event Gateway's instantiate attribute is set to *true*, then the marker for the Event Gateway looks like a Multiple Start Event (see Figure 10-113).

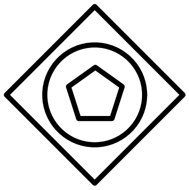


Figure 10-113 – Exclusive Event-Based Gateway to start a Process

In order for an Event Gateway to instantiate a Process, it must meet one of the following conditions:

- The Process does not have a Start Event and the Gateway has no *incoming* Sequence Flow, or
- The *incoming* Sequence Flow for the Gateway has a source of a None Start Event.
  - Note that no other *incoming* Sequence Flow are allowed for the Gateway (in particular, a loop connection from a downstream object).

In some situations a modeler may want the Process to be instantiated by one of a set of Messages while still requiring all of the Messages for the working of the same Process instance. To handle this, there is another variation of the Event Gateway.

- If the Event Gateway's instantiate attribute is set to *true* and the eventGatewayType attribute is set to Parallel, then the marker for the Event Gateway looks like a Parallel Multiple Start Event (see Figure 10-114).
    - The Event Gateway's instantiate attribute MUST be set to *true* in order for the eventGatewayType attribute to be set to Parallel (i.e., for Event Gateway's that do not instantiate the Process MUST be Exclusive—a standard Parallel Gateway can be used to include parallel Events in the middle of a Process).

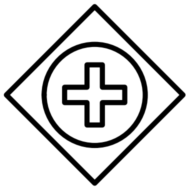


Figure 10-114 – Parallel Event-Based Gateway to start a Process

The Parallel Event Gateway is also a type of race condition. In this case, however, when the first Event is triggered and the Process is instantiated, the other Events of the Gateway configuration are not disabled. The other Events are still waiting and are expected to be triggered before the Process can (normally) complete. In this case, The Messages that trigger the Events of the Gateway configuration must share the same correlation information.

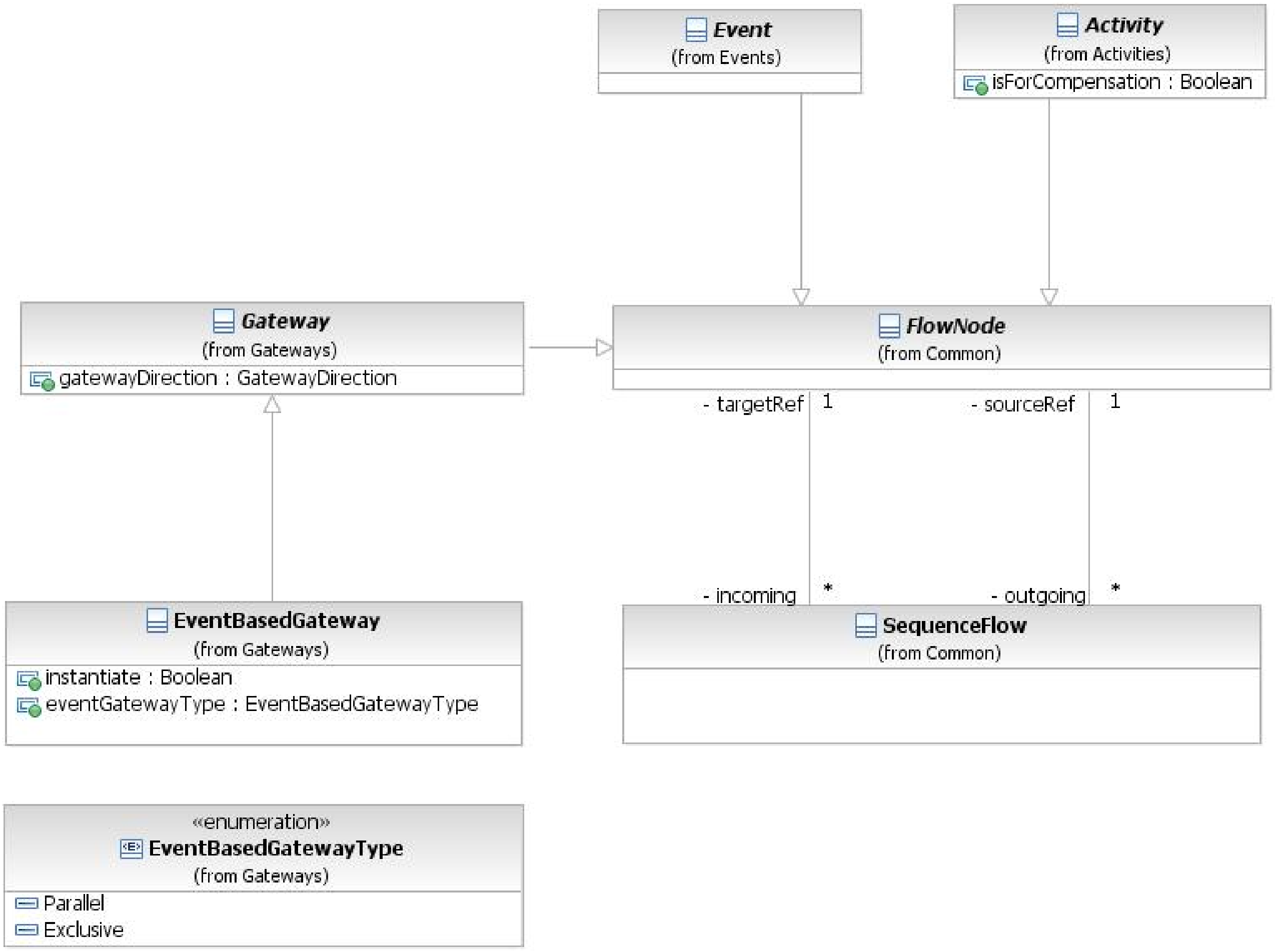


Figure 10-115 – Event-Based Gateway class diagram

The Event-Based Gateway element inherits the attributes and model associations of Gateway (see Table 8-47). Table 10-120 presents the additional attributes and model associations of the Event-Based Gateway element:

Table 10-120 – EventBasedGateway Attributes & Model Associations

Attribute Name	Description/Usage
instantiate: boolean = False	When true, receipt of one of the events will instantiate the process instance.
eventGatewayType: EventGatewayType = Exclusive { Exclusive | Parallel }	The eventGatewayType determines the behavior of the Gateway when used to instantiate a Process (as described above). The attribute can only be set to Parallel when the instantiate attribute is set to *true*.

Event-Based Gateways can be used at the start of a Process, without having to be a target of a Sequence Flow. There can be multiple such Event-Based Gateways at the start of a Process. Ordinary Start Events and Event-Based Gateways can be used together.

10.5.7. Gateway Package XML Schemas

Table 10-121 – ComplexGateway XML schema

```
<xsd:element name="complexGateway" type="tComplexGateway"
substitutionGroup="flowElement"/>
<xsd:complexType name="tComplexGateway">
        <xsd:complexContent>
                <xsd:extension base="tGateway">
                        <xsd:sequence>
                                <xsd:element name="activationCondition"  type="tExpression"
                                minOccurs="0" maxOccurs="1"/>
                        </xsd:sequence>
                        <xsd:attribute name="default" type="xsd:IDREF"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-122 – EventBasedGateway XML schema

```
<xsd:element name="eventBasedGateway" type="tEventBasedGateway"
substitutionGroup="flowElement"/>
<xsd:complexType name="tEventBasedGateway">
        <xsd:complexContent>
                <xsd:extension base="tGateway">
                        <xsd:attribute name="instantiate" type="xsd:boolean" default="false"/>
                        <xsd:attribute name="eventGatewayType"
                        type="tEventBasedGatewayType" default="Exclusive"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tEventBasedGatewayType">
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="Exclusive"/>
                <xsd:enumeration value="Parallel"/>
        </xsd:restriction>
</xsd:simpleType>
```

Table 10-123 – ExclusiveGateway XML schema

```
<xsd:element name="exclusiveGateway" type="tExclusiveGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tExclusiveGateway">
    <xsd:complexContent>
        <xsd:extension base="tGateway">
            <xsd:attribute name="default" type="xsd:IDREF" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-124 – Gateway XML schema

```
<xsd:element name="gateway" type="tGateway" abstract="true"/>
<xsd:complexType name="tGateway">
    <xsd:complexContent>
        <xsd:extension base="tFlowElement">
            <xsd:attribute name="gatewayDirection" type="tGatewayDirection" default="unspecified"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="tGatewayDirection">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="unspecified"/>
        <xsd:enumeration value="converging"/>
        <xsd:enumeration value="diverging"/>
        <xsd:enumeration value="mixed"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 10-125 – InclusiveGateway XML schema

```
<xsd:element name="inclusiveGateway" type="tInclusiveGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tInclusiveGateway">
    <xsd:complexContent>
        <xsd:extension base="tGateway">
            <xsd:attribute name="default" type="xsd:IDREF" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-126 – ParallelGateway XML schema

```
<xsd:element name="parallelGateway" type="tParallelGateway" substitutionGroup="flowElement"/>
<xsd:complexType name="tParallelGateway">
    <xsd:complexContent>
        <xsd:extension base="tGateway"/>
    </xsd:complexContent>
</xsd:complexType>
```

10.6. Compensation

Compensation is concerned with undoing steps that were already successfully completed, because their results and possibly side effects are no longer desired and need to be reversed. If an Activity is still active, it cannot be compensated, but rather needs to be canceled. Cancellation in turn may result in *compensation* of already successfully completed portions of an active Activity, in case of a Sub-Process.

Compensation is performed by a *compensation handler*. A *compensation handler* performs the steps necessary to reverse the effects of an Activity. In case of a Sub-Process, the *compensation handler* has access to Sub-Process data at the time of its completion ("snapshot data").

Compensation is triggered by a *throw* Compensation Event, which typically will be raised by an *error handler*, as part of cancellation, or recursively by another *compensation handler*. That Event specifies the Activity for which *compensation* is to be performed, either explicitly or implicitly

10.6.1. Compensation Handler

A *compensation handler* is a set of Activities that are not connected to other portions of the BPMN model. The *compensation handler* starts with a *catch* Compensation Event. That *catch* Compensation Event either is a boundary Event, or, in case of a Compensation Event Sub-Process, the *handler's* Start Event.

A *compensation handler* connected via a boundary Event can only perform "black-box" *compensation* of the original Activity. This *compensation* is modeled with a specialized Compensation Activity, which is connected to the boundary Event through an Association (see Figure 10-116). The Compensation Activity, which can be either a Task or a Sub-Process, has a marker to show that it is used for *compensation* only and is outside the *normal flow* of the Process.

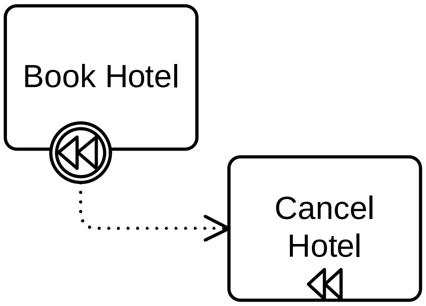


Figure 10-116 – *Compensation* through a boundary Event

A Compensation Event Sub-Process is contained within a Process or a Sub-Process (see Figure 10-117). Like the Compensation Activity, the Compensation Event Sub-Process is outside the *normal flow* of the Process. The Event Sub-Process, which is marked with a dotted line boundary, can access data that is part of its parent, a snapshot at the point in time when its parent completed. A Compensation Event Sub-Process can recursively trigger *compensation* for Activities contained in its parent.

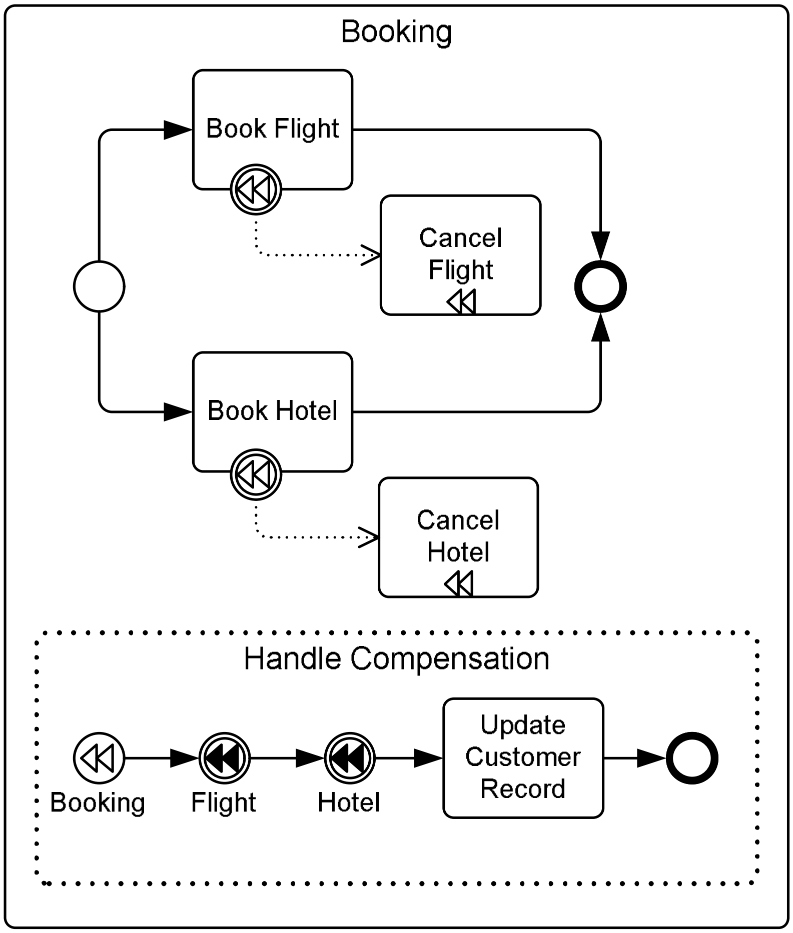


Figure 10-117 – Monitoring Class Diagram

It is possible to specify that a Sub-Process can be compensated without having to define the *compensation handler*. The Sub-Process attribute compensable, when set, specifies that default *compensation* is implicitly defined, which recursively compensates all successfully completed Activities within that Sub-Process.

The example on page 286 contains a custom Compensation Event Sub-Process, triggered by a Compensation Start Event. Note that this *compensation handler* deviates from default *compensation* in that it runs Compensation Activities in an order different from the order in the forward case; it also contains an additional Activity adding Process logic that cannot be derived from the body of the Sub-Process itself.

10.6.2. Compensation Triggering

Compensation is triggered using a *compensation throw* Event, which can either be an Intermediate or an End Event. The Activity which needs to be compensated is referenced. If the Activity is clear from the context, it doesn't have to be specified and defaults to the current Activity. A typical scenario for that is an inline *error handler* of a Sub-Process that cannot recover the *error*, and as a result would trigger *compensation* for that Sub-Process. If no Activity is specified in a "global" context, all completed Activities in the Process are compensated.

By default, *compensation* is triggered synchronously, that is, the *compensation throw* Event waits for the completion of the triggered *compensation handler*. Alternatively, *compensation* can just be triggered without waiting for its completion, by setting the *throw* Compensation Event's waitForCompletion attribute to *false*.

Multiple *instances* typically exist for Loop or Multi-Instance Sub-Processes. Each of these has its own *instance* of its Compensation Event Sub-Process, which has access to the specific snapshot data that was current at the time of completion of that particular *instance*. Triggering *compensation* for the Multi-Instance Sub-Process individually triggers *compensation* for all *instances* within the current *scope*. If *compensation* is specified via a boundary *compensation handler*, this boundary *compensation handler* also is invoked once for each *instance* of the Multi-Instance Sub-Process in the current *scope*.

10.6.3. Relationship between Error Handling and Compensation

The following items define the relationship between *error handling* and *compensation*:

- *Compensation* employs a "presumed abort principle", with the following consequences: *Compensation* of a failed Activity results in a null operation.
- When an Activity fails, i.e., is left because an *error* has been thrown, it's the *error handlers* responsibility to ensure that no further *compensation* will be necessary once the *error handler* has completed.
- If no *error* Event Sub-Process is specified for a particular Sub-Process and a particular *error*, the default behavior is to automatically call *compensation* for all contained Activities of that Sub-Process if that *error* is thrown, ensuring the behavior in for *auditing* and *monitoring*.

10.7. Lanes

A Lane is a sub-partition within a Pool or a Process and will extend the entire length of the Diagram, either vertically (see Figure 10-118) or horizontally (see Figure 10-119). If the Process is invisibly bounded, the Lane must extend the entire length of the Process. Text associated with the Lane (e.g., its name and/or that of any Process element attribute) can be placed inside the shape, in any direction or location, depending on the preference of the modeler or modeling tool vendor. Our examples place the name as a banner on the left side (for horizontal Pools) or at the top (for vertical Pools) on the other side of the line that separates the Pool name, however, this is not a requirement.

- ◆ A Lane is a square-cornered rectangle that MUST be drawn with a solid single line (see Figure 10-118 and Figure 10-119).
  - ◆ The label for the Pool MAY be placed in any location and direction within the Pool, but MUST NOT be separated from the contents of the Pool by a single line (except in the case that there are sub-Lanes within the Lane).

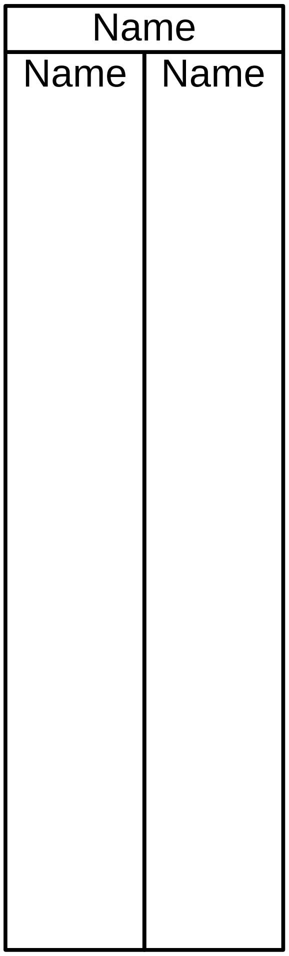


Figure 10-118 – Two Lanes in a Vertical Pool

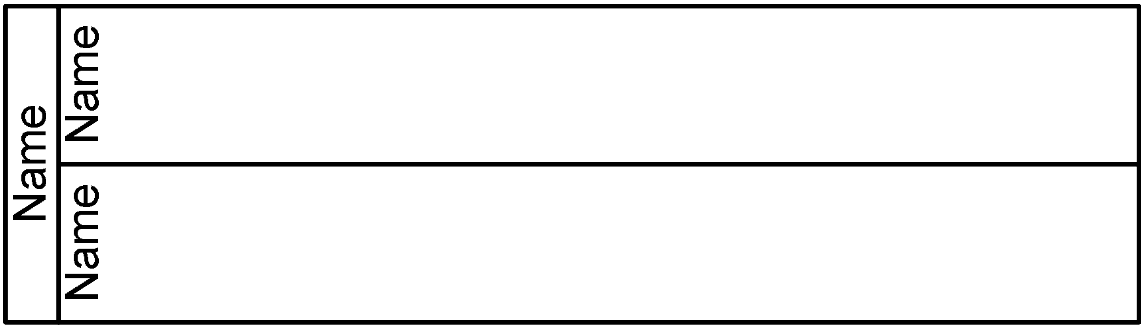


Figure 10-119 – Two Lanes in a horizontal Pool

Lanes are used to organize and categorize Activities within a Pool. The meaning of the Lanes is up to the modeler. BPMN does not specify the usage of Lanes. Lanes are often used for such things as internal roles (e.g., Manager, Associate), systems (e.g., an enterprise application), an internal department (e.g., shipping, finance), etc. In addition, Lanes can be nested (see Figure 10-120) or defined in a matrix. For example, there could be an outer set of Lanes for company departments and then an inner set of Lanes for roles within each department.

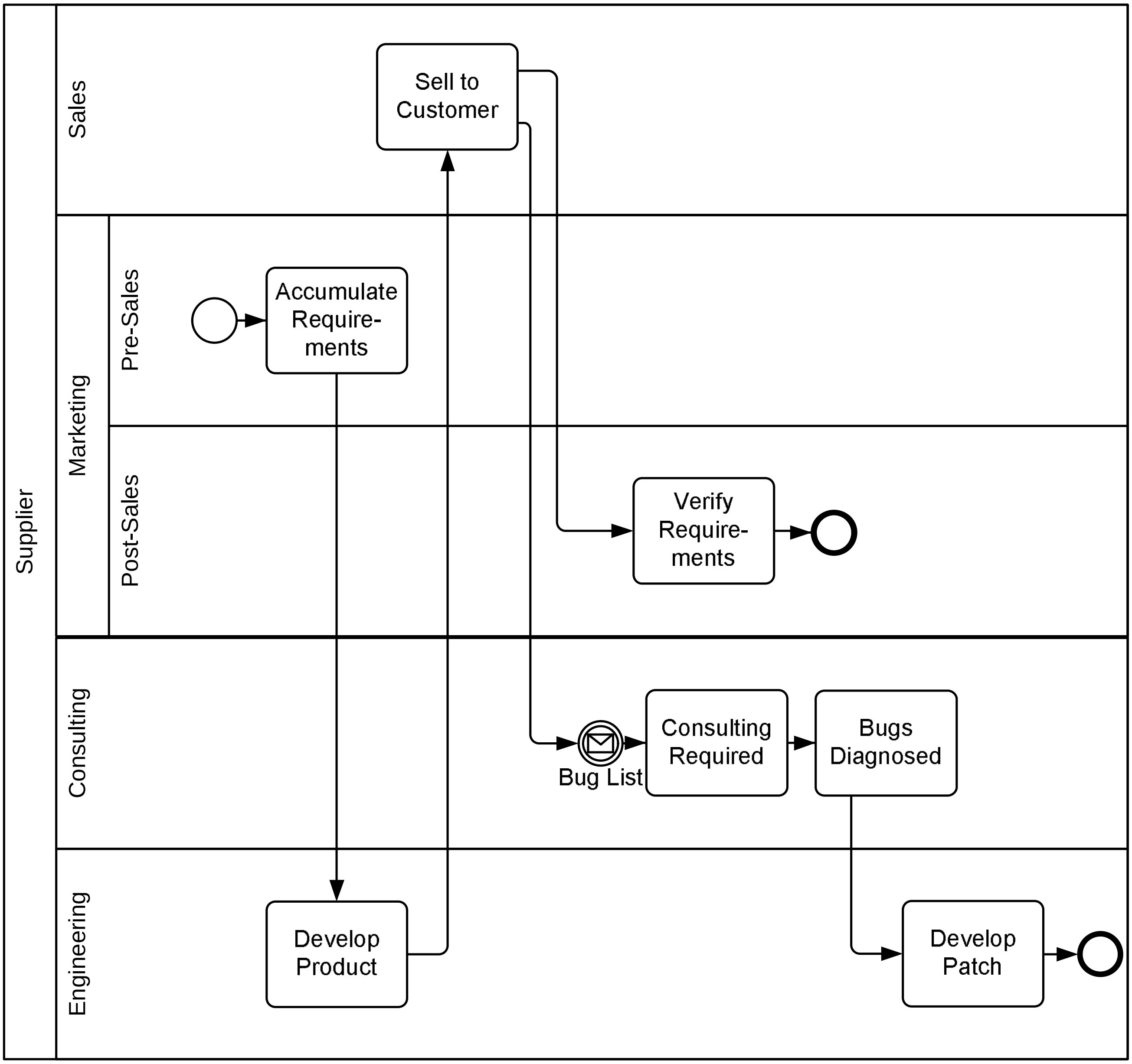


Figure 10-120 – An Example of Nested Lanes

Figure 10-121 shows the `Lane` class diagram. When a `Lane` is defined it is contained within a `LaneSet`, which is contained within a `Process`.

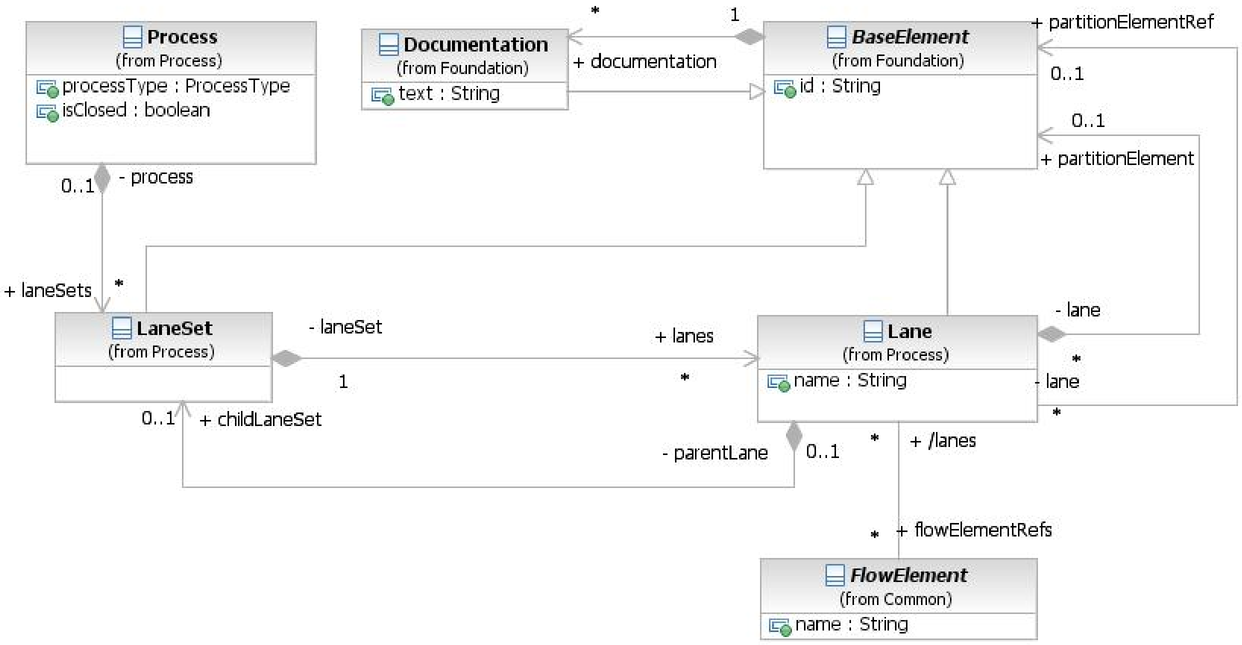


Figure 10-121 – The Lane class diagram

The LaneSet element defines the container for one or more Lanes. A Process can contain one or more LaneSets. Each LaneSet and its Lanes can partition the *Flow Elements* in a different way.

The LaneSet element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-127 presents the additional attributes and model associations of the LaneSet element:

Table 10-127 – LaneSet attributes and model associations

Attribute Name	Description/Usage
process: Process	The process owning the LaneSet
lanes: Lane [0..*]	One or more Lane elements, which define a specific partition in the LaneSet
parentLane: Lane [0..1]	The reference to a Lane element which is the parent of this LaneSet.

A Lane element defines one specific partition in a LaneSet. The Lane can define a partition element which specifies the value and element type, a tool can use to determine the list of *Flow Elements* to be partitioned into this Lane. All Lanes in a single LaneSet must define partition element of the same type, e.g. all Lanes in a LaneSet defines the Performer element as the partition element, but all with different values.

The Lane element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 10-128 presents the additional attributes and model associations of the Lane element:

Table 10-128 – Lane attributes and model associations

Attribute Name	Description/Usage
name: string	The name of the Lane
partitionElement: BaseElement [0..1]	A reference to a BaseElement which specify the partition value and partition type. Using this partition element a BPMN compliant tool can determine the FlowElements which have to be partitioned in this Lane.
partitionElementRef: BaseElement [0..1]	A reference to a BaseElement which specify the partition value and partition type. Using this partition element a BPMN compliant tool can determine the FlowElements which have to be partitioned in this Lane.
childLaneSet: LaneSet [0..1]	A reference to a LaneSet element for embedded Lanes.
flowElementRefs: FlowElement [0..*]	The list of FlowElement partitioned into this Lane according to the partitionElement defined as part of the Lane element

10.8. Process Instances, Unmodeled Activities, and Public Processes

A Process can be executed or performed many times, but each time is expected to follow the steps laid out in the Process model. For example, the Process in Figure 10-1 will occur every Friday, but each *instance* is expected to perform Task "Receive Issue List," then Task "Review Issue List," and so on, as specified in the model. Each *instance* of a Process is expected to be valid for the model, but some *instances* might not, for example if the Process has manual Activities, and the performers have not had proper instruction on how to carry out the Process.

In some applications it is useful to allow more Activities and Events to occur when a Process is executed or performed than are contained the Process model. This enables other steps to be taken as needed without changing the Process. For example, instances of the Process in Figure 10-1 might execute or perform an extra Activity between Task "Receive Issue List" and Task "Review Issue List." These *instances* are still valid for the Process model in Figure 10-1, because the *instances* still execute or perform the Activities in the Process, in the order they are modeled and under conditions specified for them.

There are two ways to specify whether unmodeled Activities are allowed to occur in Process instances:

- If the isClosed attribute of a Process has a value of *false* or no value, then interactions, such as sending and receiving Messages and Events, MAY occur in an *instance* without additional flow elements in the Process. Unmodeled interactions can still be restricted on particular Sequence Flow in the Process (see next bullet). If the isClosed attribute of a Process has a value of *true*, then interactions, such as sending and receiving Messages and Events, MAY NOT occur without additional flow elements in the Process. This restriction overrides any unmodeled interactions allowed by Sequence Flow in the next bullet.

- If the isImmediate attribute of a Sequence Flow in a Process has a value of *false*, then other Activities and interactions not modeled in the Process MAY be executed or performed during the Sequence Flow. If the isImmediate attribute has a value of *true*, then Activities and interactions not modeled in the Process MAY NOT be executed or performed during Sequence Flow. In *public* Processes (processType attribute has value public) Sequence Flow with no value for isImmediate are treated as if the value were *false*. In *private* Processes (processType attribute has value executable or non-executable) Sequence Flow with no value for isImmediate are treated as if the value were *true*. *Executable* Processes cannot have a *false* value for the isImmediate attribute.

Restrictions on unmodeled Activities specified with isClosed and isImmediate apply only under executions or performances (*instances*) of the Process containing the restriction. These Activities MAY occur in *instances* of other Processes.

When a Process allows Activities to occur that the Process does not model, those Activities might appear in other Process models. The executions or performances (*instances*) of these other Processes might be valid for the original Process. For example, a Process might be defined similar to the one in Figure 10-1 that adds an extra Activity between Task "Receive Issue List" and Task "Review Issue List." The Process in Figure 10-1 might use isClosed or isImmediate to allow other Activities to occur in between Task "Receive Issue List" and Task "Review Issue List." When the Process is executed or performed, then *instances* of the other Process (the one with the extra step in between Task "Receive Issue List" and Task "Review Issue List") will be valid for the Process in Figure 10-1. Modelers can declare that they intend all *instances* of one Process will be valid for another Process using the supports association between the Processes. During development of these Processes, support might not actually hold, because the association just expresses modeler intent.

A common use for model support is between *private* and *public* Processes, see Section "Overview" (page 40). A *public* Process contain Activities visible to external parties, such as *Participants* in a Collaboration, while a *private* Process includes other Activities that are not visible to external parties. The hidden Activities in a *private* Process are not modeled in the *public* Process. However, it is expected that *instances* of the *private* Process will appear to external parties as if they could be *instances* of the *public* Process. This means the *private* Process supports the *public* Process (it is expected that all *instances* of the *private* Process will be valid for the *public* one).

A Process that supports another, as a *private* Process can to a *public* Process, does not need to be entirely similar to the other Process. It is only required that *instances* of the Process appear as if they could be *instance* of the other Process. For example Figure 10-122 shows a *public* Process at the top with a Send Task and Receive Task. A supporting *private* Process is shown at the bottom. The *private* Process sends and receives the same Messages, but using Events instead of Tasks. It also introduces an Activity not modeled in the *public* Process. However all *instances* of the *private* Process will appear as if they could be *instances* of the *public* one, because the Messages are sent and received in the order required by the public Process, and the *public* Process allows unmodeled Activities to occur.

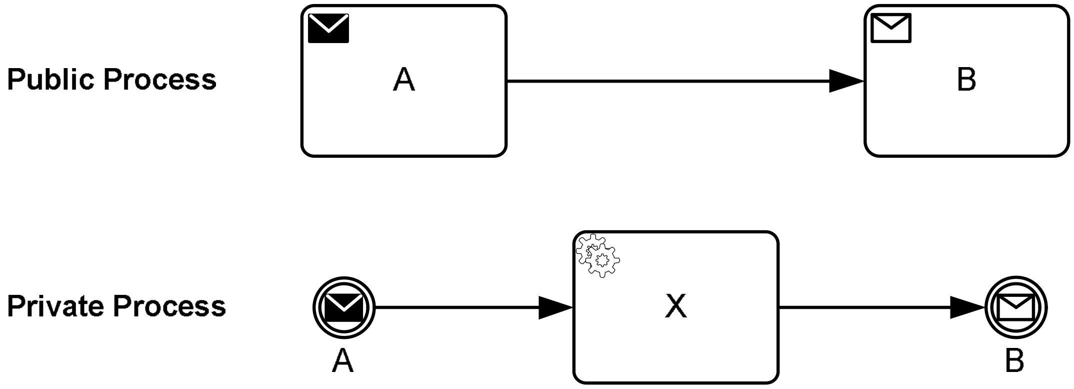


Figure 10-122 – One Process supporting to another

In practice, a *public* Process looks like an underspecified *private* Process. Anything not specified in the *public* Process is determined by the *private* one. For example, if none of the *outgoing* Sequence Flow for an Exclusive Gateway have conditions, the *private* Process will determine which one of the Activities targeted by the Sequence Flow will occur. Another example is a Timer Event with no EventDefinition. The *private* Process will determine when the timer goes off.

10.9. Auditing

The Auditing element and its model associations allow defining attributes related to auditing. It leverages the BPMN extensibility mechanism. This element is used by FlowElements and Process. The actual definition of auditing attributes is out of scope of this specification. BPMN 2.0 implementations may define their own set of attributes and their intended semantics.

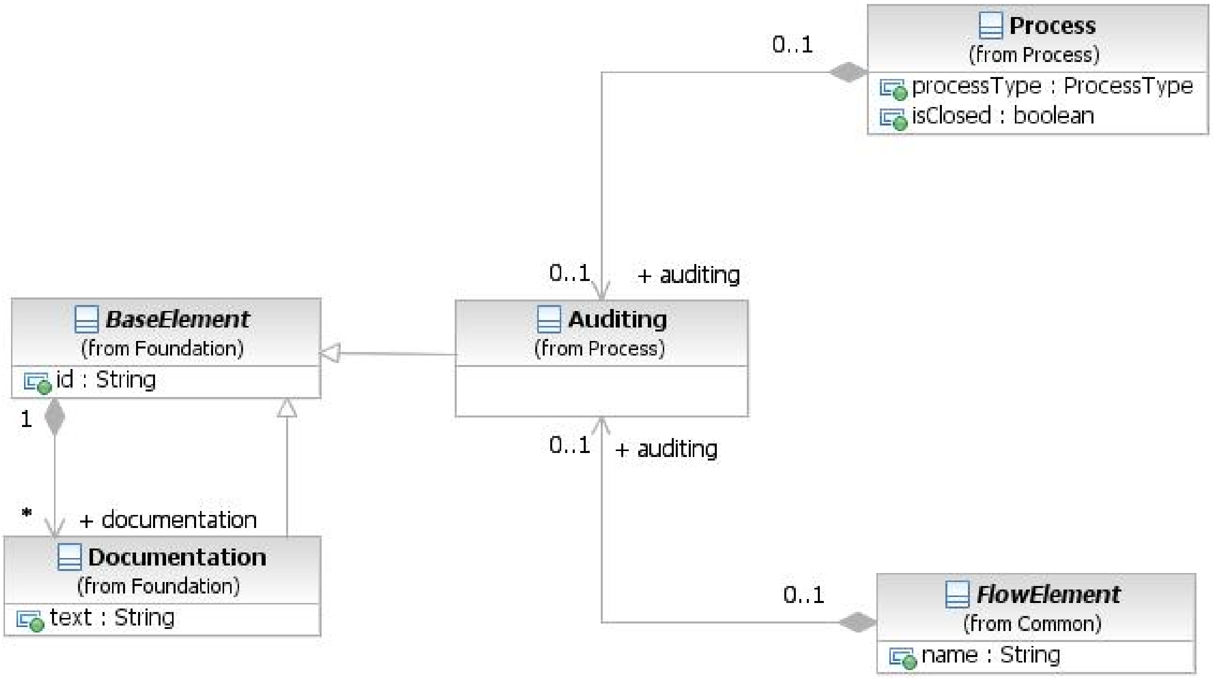


Figure 10-123 – Auditing Class Diagram

10.10.Monitoring

The Monitoring and its model associations allow defining attributes related to monitoring. It leverages the BPMN extensibility mechanism. This element is used by FlowElements and Process. The actual definition of monitoring attributes is out of scope of this specification. BPMN 2.0 implementations may define their own set of attributes and their intended semantics.

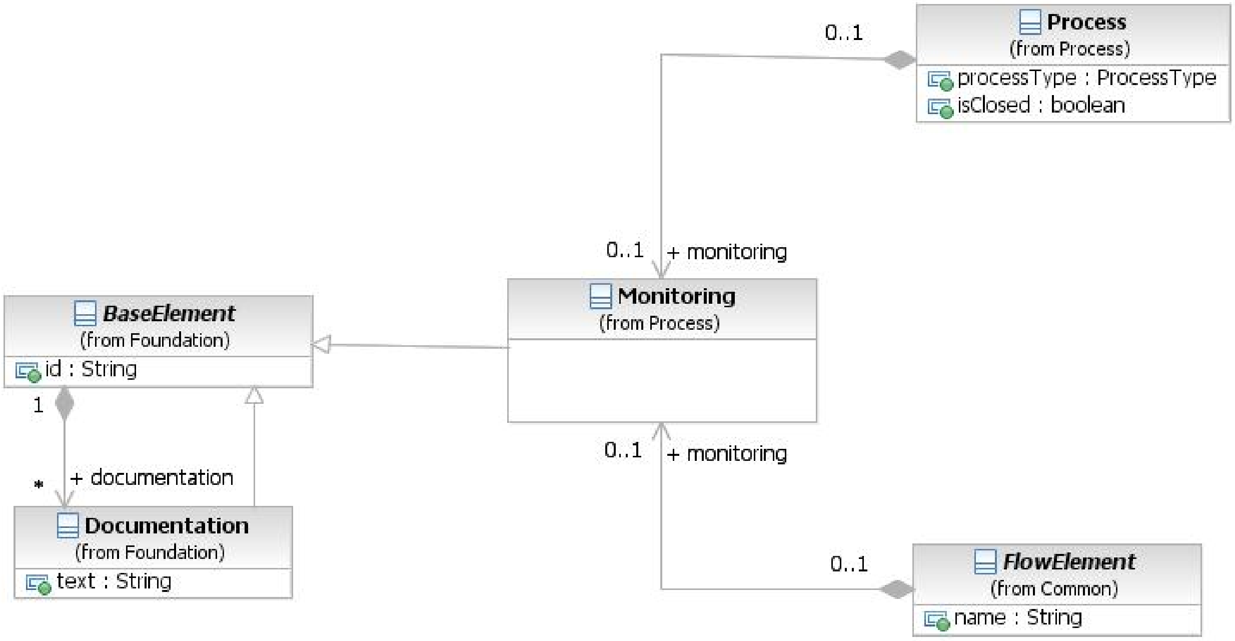


Figure 10-124 – Monitoring Class Diagram

10.11.Process within Collaboration

10.12.Process Package XML Schemas

Table 10-129 – Process XML schema

```
<xsd:element name="process" type="tProcess" substitutionGroup="rootElement"/>
<xsd:complexType name="tProcess">
        <xsd:complexContent>
                <xsd:extension base="tCallableElement">
                        <xsd:sequence>
                                <xsd:element ref="auditing" minOccurs="0" maxOccurs="1"/>
                                <xsd:element ref="monitoring" minOccurs="0" maxOccurs="1"/>
                                <xsd:element ref="processRole" minOccurs="0"
                                maxOccurs="unbounded"/>
                                <xsd:element ref="property" minOccurs="0"
                                maxOccurs="unbounded"/>
                                <xsd:element ref="laneSet" minOccurs="0"
                                maxOccurs="unbounded"/>
                                <xsd:element ref="flowElement" minOccurs="0"
                                maxOccurs="unbounded"/>
                                <xsd:element ref="artifact" minOccurs="0"
                                maxOccurs="unbounded"/>
                                <xsd:element name="supports" type="xsd:QName" minOccurs="0"
                                maxOccurs="unbounded"/>
                        </xsd:sequence>
                        <xsd:attribute name="processType" type="tProcessType" default="none"/>
                        <xsd:attribute name="isClosed" type="xsd:boolean" default="false"/>
                        <xsd:attribute name="definitionalCollaborationRef" type="xsd:QName"
                        use="optional"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="tProcessType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="none"/>
        <xsd:enumeration value="public"/>
        <xsd:enumeration value="executable"/>
        <xsd:enumeration value="non-executable"/>
    </xsd:restriction>
</xsd:simpleType>
```

Table 10-130 – Auditing XML schema

```
<xsd:element name="auditing" type="tAuditing"/>
<xsd:complexType name="tAuditing">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-131 – GlobalTask XML schema

```
<xsd:element name="globalTask" type="tGlobalTask" substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalTask">
        <xsd:complexContent>
                <xsd:extension base="tCallableElement">
                        <xsd:sequence>
                                <xsd:element ref="performer" minOccurs="0"
                                maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-132 – Lane XML schema

```
<xsd:element name="lane" type="tLane"/>
<xsd:complexType name="tLane">
        <xsd:complexContent>
                <xsd:extension base="tBaseElement">
                        <xsd:sequence>
                                <xsd:element name="partitionElement" type="tBaseElement"
                                minOccurs="0" maxOccurs="1"/>
                                <xsd:element name="flowElementRef" type="xsd:IDREF"
                                minOccurs="0" maxOccurs="unbounded"/>
                                <xsd:element name="childLaneSet" type="tLaneSet"
                                minOccurs="0" maxOccurs="1"/>
                        </xsd:sequence>
                        <xsd:attribute name="name" type="xsd:string"/>
                        <xsd:attribute name="partitionElementRef" type="xsd:IDREF"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-133 – LaneSet XML schema

```
<xsd:element name="laneSet" type="tLaneSet"/>
<xsd:complexType name="tLaneSet">
        <xsd:complexContent>
                <xsd:extension base="tBaseElement">
                        <xsd:sequence>
                                <xsd:element ref="lane" minOccurs="0"
                                maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

Table 10-134 – Monitoring XML schema

```
<xsd:element name="monitoring" type="tMonitoring"/>
<xsd:complexType name="tMonitoring">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement"/>
    </xsd:complexContent>
</xsd:complexType>
```

Table 10-135 – Performer XML schema

```
<xsd:element name="performer" type="tPerformer" substitutionGroup="activityResource"/>
<xsd:complexType name="tPerformer">
        <xsd:complexContent>
                <xsd:extension base="tActivityResource"/>
        </xsd:complexContent>
</xsd:complexType>
```

11. Conversation

The Conversation diagram is similar to a Collaboration diagram. However, the Pools of a Conversation are not allowed to contain a Process and a Choreography is not allowed to be placed in between the Pools of a Conversation diagram.

The view includes two (2) additional graphical elements that do not exist in other BPMN views:

- A Communication
- A CommunicationLink

A Conversation is set of Message exchanges (Message Flow) that share the same *Correlation*.

A Conversation is the logical relation of Message exchanges. The logical relation, in practice, often concerns a business object(s) of interest, e.g. “Order,” “Shipment and Delivery,” and “Invoice.” Hence, a Conversation is associated with a set of name-value pairs, or a Correlation Key (e.g. “Order Identifier,” “Delivery Identifier”), which is recorded in Messages that are exchanged. In this way, a Message can be routed to the specific Process *instance* responsible for receiving and processing the Message.

Figure 11-1 shows a simple example of a Conversation diagram

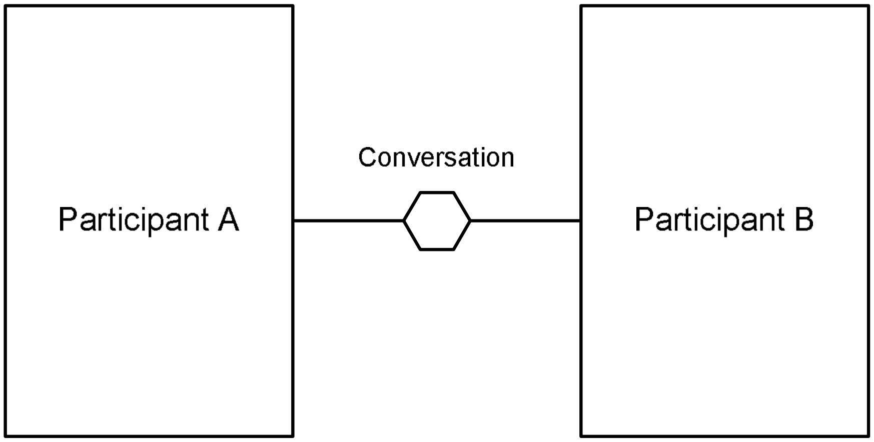


Figure 11-1 – A Conversation diagram

Figure 11-2 shows a variation of the example above where the Conversation node has been expanded into its component Message Flow.

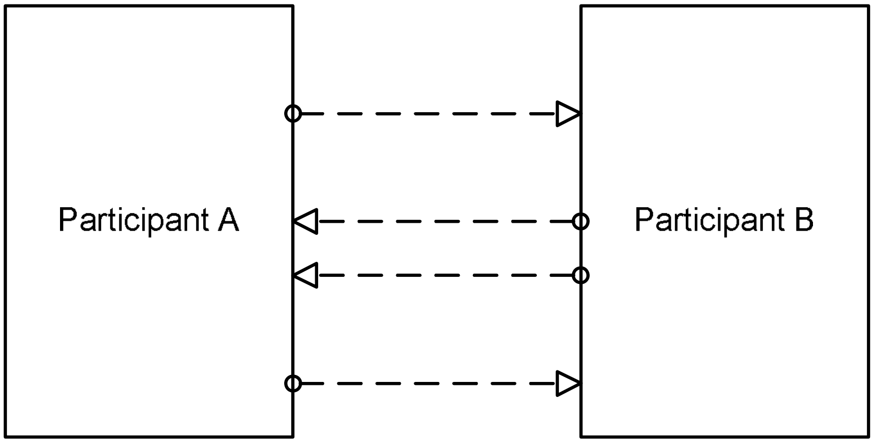


Figure 11-2 – A Conversation diagram where the Conversation is expanded into Message Flow

Message exchanges are related to each other and reflect distinct business scenarios. The relation is sometimes simple, e.g. a request followed by a response (and can be described as part of a structural interface of a service, e.g. as a WSDL operation definition). However for commercial business transactions managed through Business Processes, the relation can be complex, involving long-running, reciprocal Message exchanges, and that could extend beyond bilateral to complex, multilateral Collaborations. For example, in logistics, stock replenishments involve the following types scenarios: creation of sales orders; assignment of carriers for shipments combining different sales orders; crossing customs/quarantine; processing payment and investigating exceptions.

In addition to an *orchestration* Process, Conversations are relevant to a Choreography. The difference is that a Choreography provides a multi-party perspective of a Conversation. This is because the Message exchanges modeled using Choreography Activities concern multiple Participants, unlike an *orchestration* Process where the Message sending and receiving elements relate to one Participant only. Other than the difference in perspective, the notion of Conversation remains the same across Choreography and *orchestration* - and the Message exchanges of a Conversation will ultimately to be executed through an *orchestration* Process.

Since Choreography provides a top-down, design-time modeling perspective for Message exchanges and their Conversations, an abstracted view of the all Conversations pertaining to a domain being modeled is available through a Conversation diagram. A Conversation diagram, as depicted in Figure 11-3, shows Conversations (as hexagons) between *Participants*. This provides a "bird's eye" perspective of the different Conversations which relate to the domain.

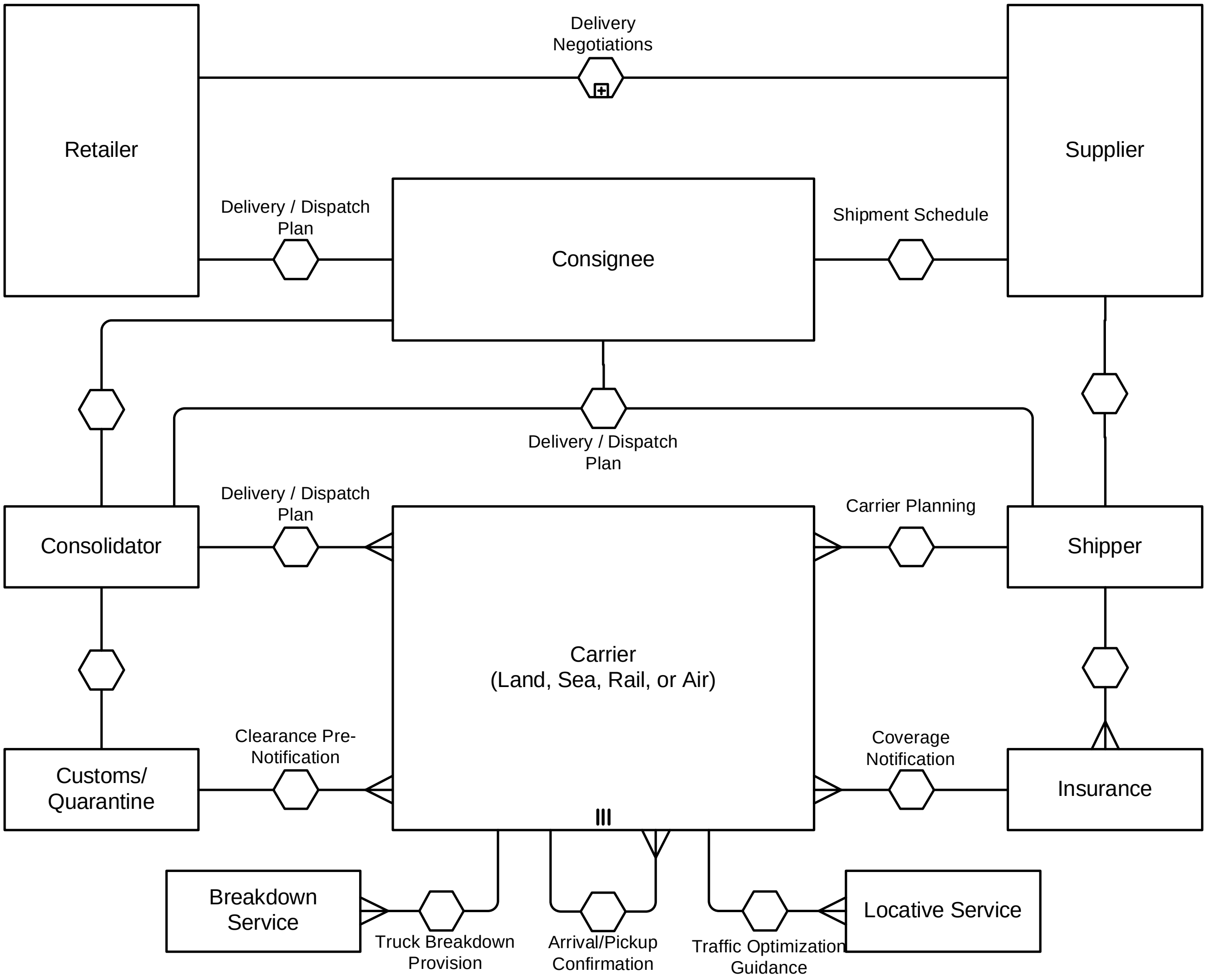


Figure 11-3 – Conversation **diagram depicting several conversations between *Participants* in a related domain**

Figure 11-3, above, depicts 13 distinct Conversations between collaborating Participants in a logistics domain. As examples, *Retailer* and *Supplier* are involved in a *Delivery Negotiations* Conversation, and *Consignee* converses with *Retailer* and *Supplier* through *Delivery/Dispatch Plan* and *Shipment Schedule* Conversations respectively. More than two participants may be involved in a Conversation, e.g. *Consignee, Consolidator* and *Shipper* in *Detailed Shipment Schedule*. The association of Participants to a Conversation are constrained to indicate whether one or many of *Participants* are involved. For example, one *instance* of *Retailer* converses with one *instance* of *Supplier* for *Deliver Negotiations*. However, one *instance* of *Shipper* converses with multiple *instances* of *Carrier* (indicated by the multiplicity symbol "*" near *Carrier*) for *Carrier Planning*. Note, multiplicity in constraints of Conversation diagrams means one or more (not zero or more).

The behavior of different Conversations is modeled through separate Choreographies, detailing the Message exchange sequences. In practice, Conversations which are closely related could be combined in the same Choreography models – e.g. a Message exchange in the *Delivery Negotiation* leads to *Shipment Schedule, Delivery Planning* and *Delivery/Dispatch* Conversations and these could be combined together in the same Choreography. Alternatively, they could be separated in different models.

Figure 11-4 shows how Message exchanges can be expanded from a Conversation in the Conversation diagram of Figure 11-3, above. This expands the Conversation with the Message Flow, providing a structural view of a Conversation without the "clutter" of sequencing details in the same diagram. Figure 11-4 also indicates the Correlation Key involved in the Message Flow of the Conversation. For example, *Order Id* is required for in all Messages of Message Flow in Delivery Negotiation. In addition, some Message Flow also require *Variation Id* (for dealing with shipment variations on a per line item basis).

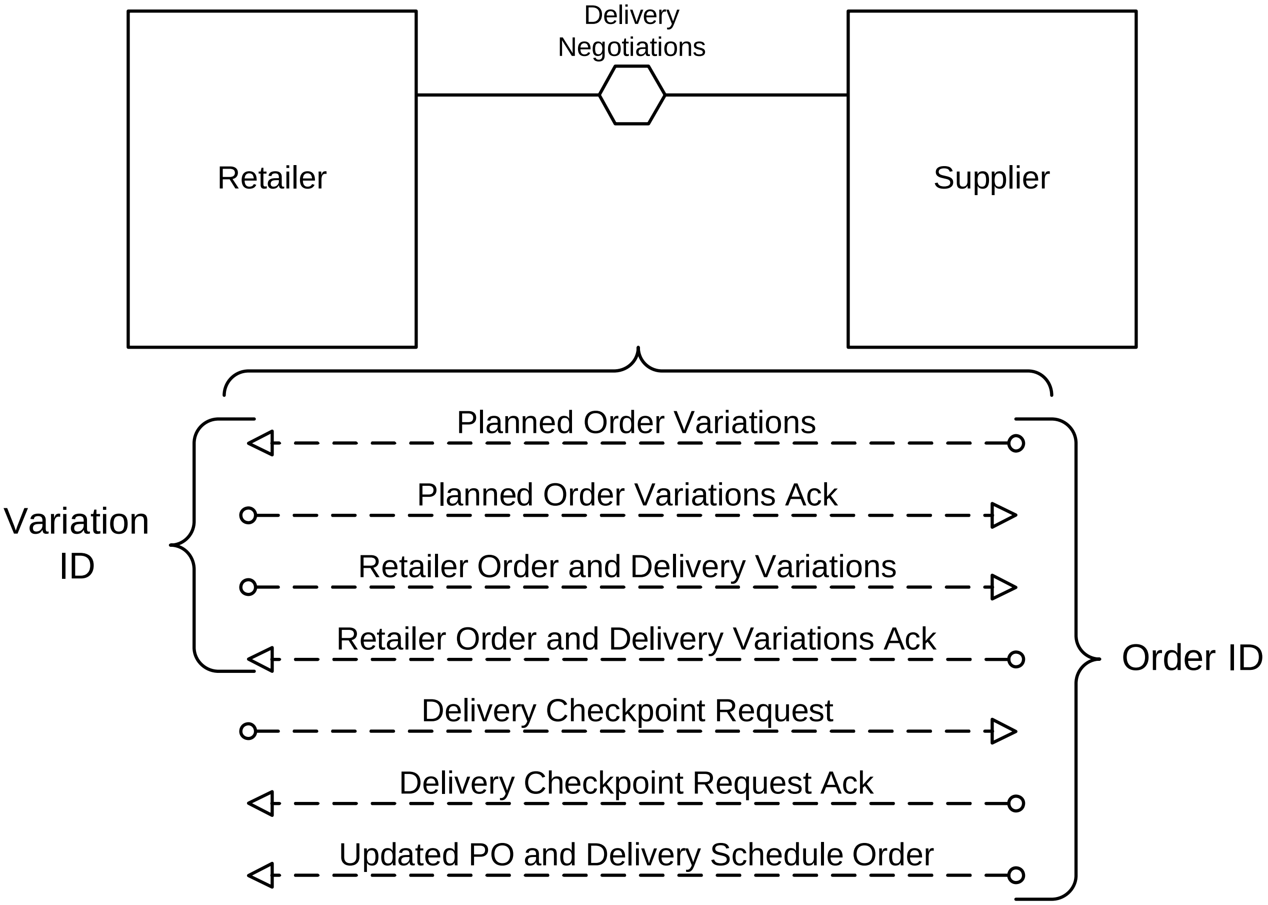

Delivery Negotiations
Retailer
Supplier
Planned Order Variations
Planned Order Variations Ack
Retailer Order and Delivery Variations
Retailer Order and Delivery Variations Ack
Delivery Checkpoint Request
Delivery Checkpoint Request Ack
Updated PO and Delivery Schedule Order
Variation ID
Order ID

Figure 11-4 – Conversational view choreographies

In Figure 11-3, above, a hierarchical structure of Conversations can be seen with one set of Message Flow occurring within another in a parent-child relationship. In particular, after Planned Order Variations (keyed on Order Id) at the parent, a number of Message Flow of the child follow till *Retailer Order and Delivery Variations Ack* (keyed on *Variable Id* and *Order Id*). The remaining Message Flow (keyed on *Order Id*) are at the parent level. The child Conversation, as such, is part of the parent Conversation. Nesting is indicated graphically on a Conversation symbol (by a "+") to alert the modeler that one of more Conversations can take place inside the Conversation exposed in the Conversation diagram. Nesting can go to an arbitrary number of levels.

A common dependency between Conversations is overlap. Overlap occurs when two or more Conversations have some Message exchanges in common but not others. As an example in Figure 11-3, above, a Message is sent as part of Detailed Shipment Schedule (keyed on Carrier Schedule Id) to trigger

Delivery Monitoring (keyed on Shipment Id). During Delivery Monitoring, Message could be sent to Detailed Shipment Schedule (to request modifications when transportation exceptions occur).

Splits and *joins* are special types of overlap scenarios. A Conversation *split* arises when, as part of a Conversation, a message is exchanged between two or more *Participants* that at the same time spawns a new, distinct Conversation (either between the same set of *Participants* or another set). Additionally, no further Message exchanges are shared by the split Conversations as well as no subsequent merges of them occur. An example is Delivery Planning which leads to Carrier Planning and Special Cover. A Conversation *join* occurs when several Conversations are merged into one Conversation and no further Message exchanges occur in the original Conversations, i.e. these Conversations are finalized. The generalization of a *split* and *join* is a Conversation *refactor* where Conversations are split into parallel Conversations and then are merged at a later point in time.

Figure 11-5 displays the Conversation class diagram. When a Conversation is defined it is contained within Definitions.

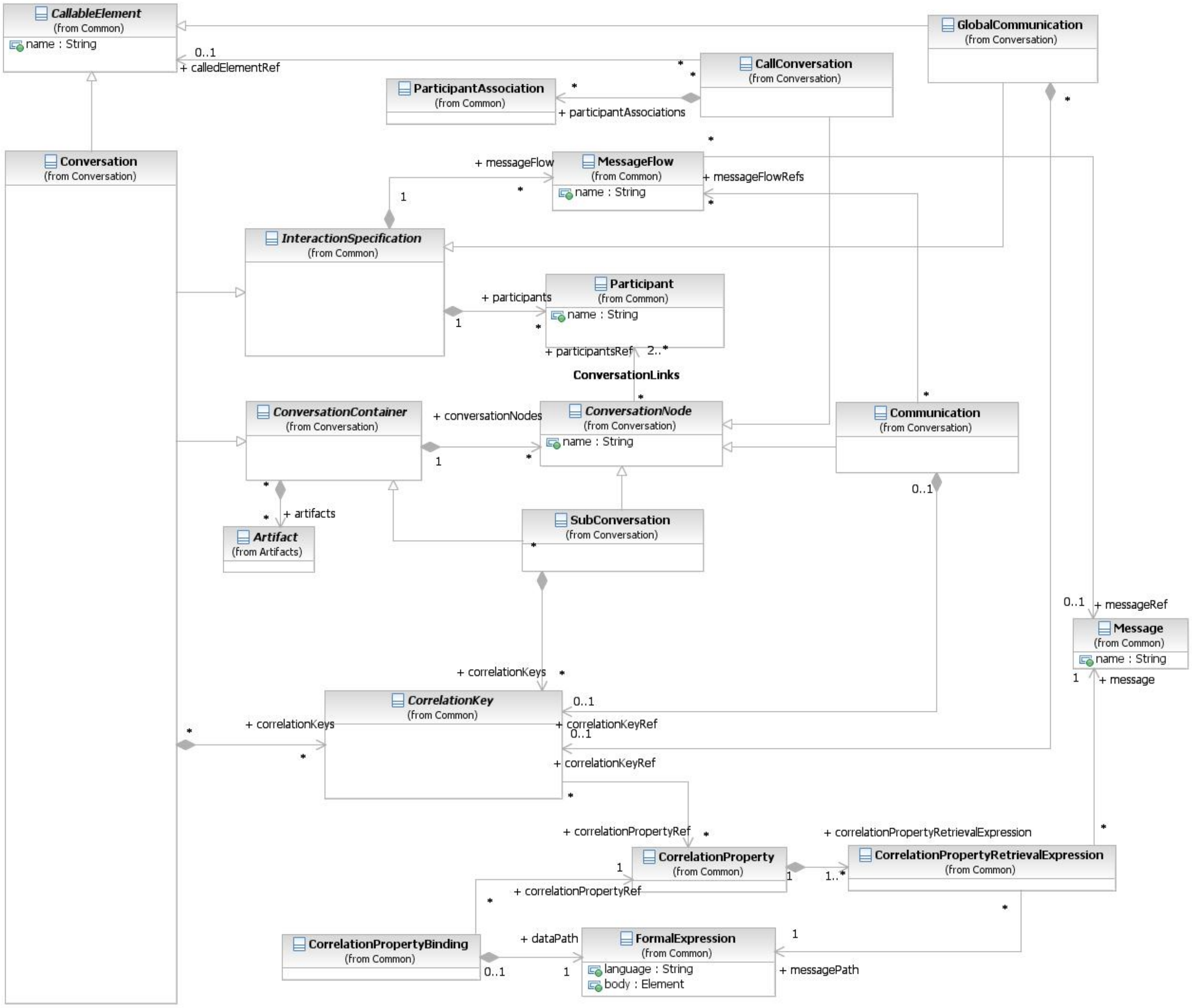


Figure 11-5 – The Conversation **Metamodel**

The Conversation element inherits the attributes and model associations of CallableElement (see Table 8-30), InteractionSpecification (see Table 8-48), and ConversationContainer (see Table 11-2). Table 11-1 presents the additional model associations for the Conversation element:

Table 11-1 – Conversation Model Associations

Attribute Name	Description/Usage
correlationKeys: CorrelationKey [0..*]	This association specifies correlationKeys used to associate Messages to a particular Conversation.
messageFlowRefs: MessageFlow [0..*]	A reference to all Message Flow (and consequently Messages) included in the Collaboration or Choreography where the Conversation is contained. This is only used when the Conversation is contained in a Choreography or Collaboration, not when the Conversation is reusable (and contained in Definitions).

Note: The CallableElement attributes supportedInterfaceRefs, ioSpecification, and ioBinding are not applicable to a Conversation.

11.1.Conversation Container

ConversationContainer is an abstract super class for the Conversation diagrams and defines the superset of elements that are contained in those diagrams. Basically, a ConversationContainer contains ConversationNodes, which are Communication (see page 336), Sub-Conversation (see page 336), and Call Conversation (see page 337).

There are two (2) types of FlowElementContainers (see Figure 11-6): Conversation and Sub-Conversation.

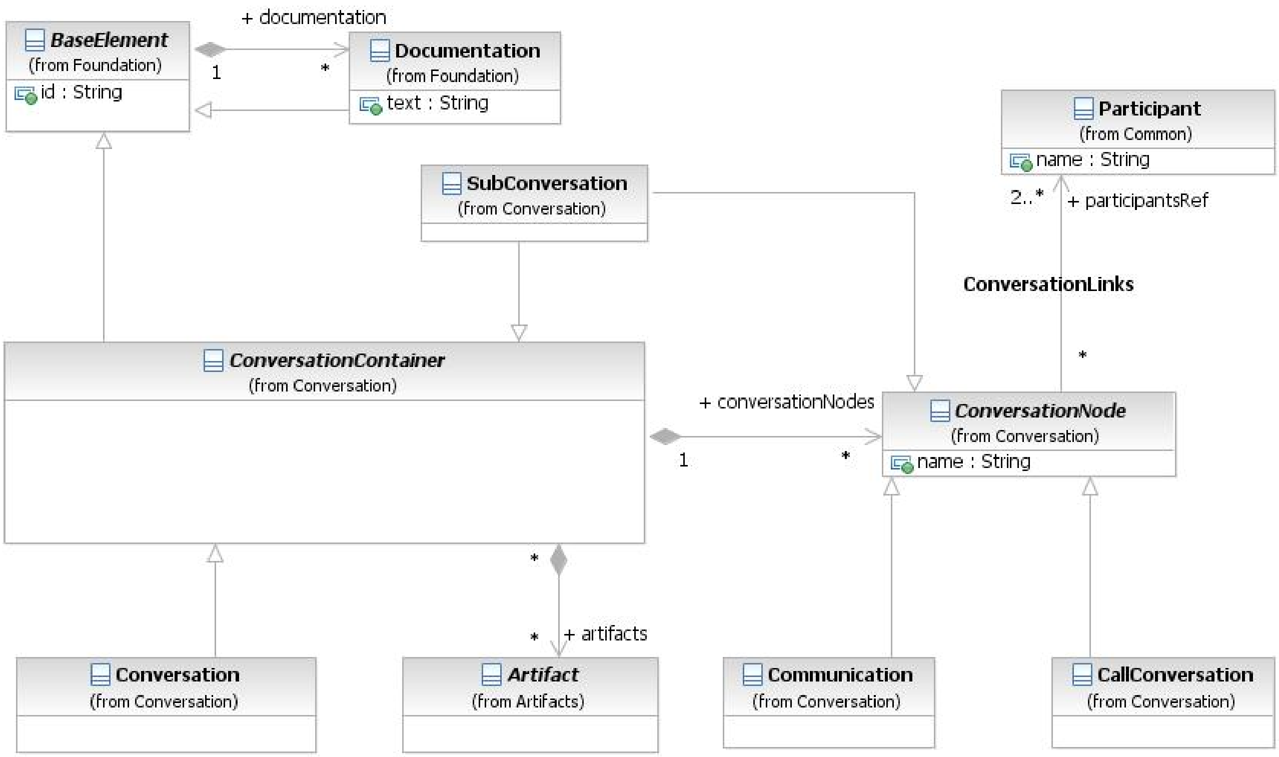


Figure 11-6 – A ConversationContainer element

The `ConversationContainer` element inherits the attributes and model associations of `BaseElement` (see Table 8-5). Table 11-2 presents the additional model associations for the `ConversationContainer` element:

Table 11-2 – ConversationContainer Model Associations

Attribute Name	Description/Usage
conversationNodes: ConversationNode [0..*]	This association specifies the particular `ConversationNodes` contained in a `ConversationContainer`. `ConversationNodes` are Communications, Sub-Conversations, and Call Conversations.
artifacts: Artifact [0..*]	This attribute provides the list of Artifacts that are contained within the `ConversationContainer`.

11.2.Conversation Node

`ConversationNode` is the abstract super class for all elements that can appear in a Conversation diagram, which are Communication (see page 336), Sub-Conversation (see page 336), and Call Conversation (see page 337).

`ConversationNodes` are linked to and from *Participants* using Communication Links (see page 338).

The ConversationNode element inherits the attributes and model associations of BaseElement (see Table 8-5). Table 11-3 presents the additional attributes and model associations for the ConversationNode element:

Table 11-3 – ConversationNode Model Associations

Attribute Name	Description/Usage
name: string [0..1]	Name is a text description of the ConversationNode.
participantRefs: Participant [0..*]	This provides the list of *Participants* that are used in the ConversationNode from the list provided by the ConversationNode's parent Conversation.

11.3. Communication

A Communication is an atomic element for a Conversation diagram. It represents a set of Message Flow grouped together based on a single CorrelationKey. A Communication will involve two (2) or more *Participants*.

- A Communication is a hexagon that MUST be drawn with a single thin line (see Figure 11-7).

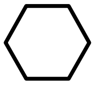


Figure 11-7 – A Communication **element**

The Communication element inherits the attributes and model associations of ConversationNode (see Table 11-3). Table 11-4 presents the additional model associations for the Communication element:

Table 11-4 – Communication Model Associations

Attribute Name	Description/Usage
correlationKeyRef: CorrelationKey [0..1]	This is a reference to one of the Conversation's correlationKeys, which is used to group Message Flow for the Communication.
messageFlowRefs: MessageFlow [0..*]	A reference to all Message Flow (and consequently Messages) included in a Communication.

11.4.Sub-Conversation

A Sub-Conversation is a ConversationNode that is a hierarchical division within the parent Conversation. A Sub-Conversation is a graphical object within a Conversation, but it also can be "opened up" to show the lower-level Conversation, which consist of Message Flow, Communications, and/or other Sub-Conversations. The Sub-Conversation shares the *Participants* of its parent Conversation.

- A Sub-Conversation is a hexagon that MUST be drawn with a single thin line (see Figure 11-8).
  - The Sub- Conversation marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the shape.

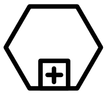


Figure 11-8 – A compound Conversation element

The Sub-Conversation element inherits the attributes and model associations of ConversationNode (see Table 11-3). Table 11-5 presents the additional model associations for the Sub-Conversation element:

Table 11-5 – Sub-Conversation Model Associations

Attribute Name	Description/Usage
correlationKeyRefs: CorrelationKeys [0..*]	This is a reference to the Conversation's correlationKeys, which are used to group Message Flow for the Sub-Conversation.

11.5.Call Conversation

A Call Conversation identifies a place in the Conversation where a global Conversation or a GlobalCommunication is used.

- If the Call Conversation calls a GlobalCommunication, then the shape will be the same as a Communication, but the boundary of the shape will MUST have a thick line (see Figure 11-9)
- If the Call Activity calls a Conversation, then the shape will be the same as a Sub-Conversation, but the boundary of the shape will MUST have a thick line (see Figure 11-10)

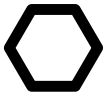


Figure 11-9 – A Call Conversation calling a GlobalCommunication

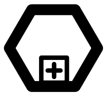


Figure 11-10 – A Call Conversation calling a Conversation

The Call Conversation element inherits the attributes and model associations of ConversationNode (see Table 11-3). Table 11-6 presents the additional model associations for the Communication element:

Table 11-6 – Communication Model Associations

Attribute Name	Description/Usage
calledElementRef: CallableElement [0..1]	The element to be called, which will be either a Conversation or a GlobalCommunication. Other CallableElements, such as Process, GlobalTask, Choreography, and GlobalChoreographyTask MUST NOT be called by the Call Conversation element.
participantAssociations: Participant Association [0..*]	This attribute provides a list of mappings from the *Participants* of a referenced GlobalCommunication or Conversation to the *Participants* of the parent Conversation.

11.6.Global Communication

A GlobalCommunication is a reusable, atomic Communication definition that can be called from within any Conversation by a Call Conversation.

The GlobalCommunication element inherits the attributes and model associations of InteractionSpecification (see Table 8-48) and CallableElement (see Table 8-30). Table 11-7 presents the additional model associations for the GlobalCommunication element:

Table 11-7 – GlobalCommunication Model Associations

Attribute Name	Description/Usage
correlationKeys: CorrelationKey [0..1]	This association specifies correlationKeys used to associate Message Flow to the GlobalCommunication.

Note: The CallableElement attributes supportedInterfaceRefs, ioSpecification, and ioBinding are not applicable to a Global Communication.

11.7.Communication Link

Conversation Links are used to connect ConversationNodes to and from *Participants* (Pools). If the target *Participant* has a participantMultiplicity of greater than one (1), then the end of the connector is forked (see Figure 11-11).

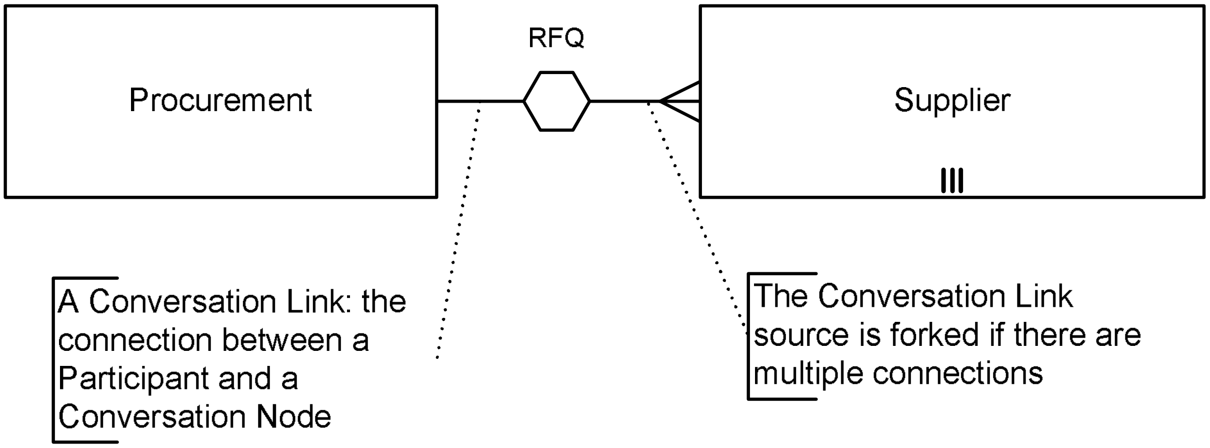

Procurement
RFQ
Supplier
A Conversation Link: the connection between a Participant and a Conversation Node
The Conversation Link source is forked if there are multiple connections

Figure 11-11 – A Conversation Link element

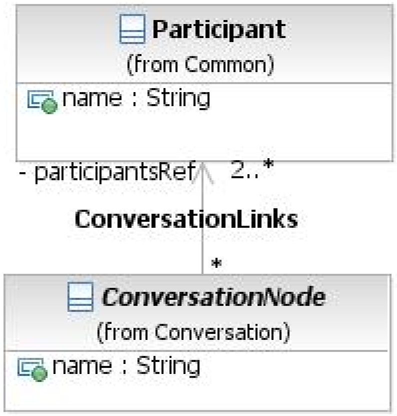


Figure 11-12 – Where Conversation Links are derived in the metamodel

There is not a specific BPMN metamodel element to represent the Conversation Link. Instead, the graphical element is derived from the relationship between ConversationNode and *Participant* (as seen in Figure 11-12).

11.8.Conversation Package XML Schemas

Table 11-8 – Call Conversation XML schema

```
<xsd:element name="callConversation" type="tCallConversation" substitutionGroup="conversationNode"/>
<xsd:complexType name="tCallConversation">
    <xsd:complexContent>
        <xsd:extension base="tConversationNode">
            <xsd:sequence>
                <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="calledElementRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11-9 – Communication XML schema

```
<xsd:element name="communication" type="tCommunication" substitutionGroup="conversationNode"/>
<xsd:complexType name="tCommunication">
    <xsd:complexContent>
        <xsd:extension base="tConversationNode">
            <xsd:sequence>
                <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name=" correlationKeyRef" type="xsd:QName" use="optional"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11-10 – Conversation XML schema

```
<xsd:element name="conversation" type="tConversation" substitutionGroup="rootElement"/>
<xsd:complexType name="tConversation">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="conversationNode" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
                <xsd:element ref="correlationKey" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11-11 – Conversation Node XML schema

```
<xsd:element name="conversationNode" type="tConversationNode"/>
<xsd:complexType name="tConversationNode" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tBaseElement">
            <xsd:sequence>
                <xsd:element name="participantRef" type="xsd:QName" minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11-12 – Global Communication XML schema

```
<xsd:element name="globalCommunication" type="tGlobalCommunication"
        substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalCommunication">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="correlationKey" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 11-13 – Sub-Conversation XML schema

```
<xsd:element name="subConversation" type="tSubConversation" substitutionGroup="conversationNode"/>
<xsd:complexType name="tSubConversation">
    <xsd:complexContent>
        <xsd:extension base="tConversationNode">
            <xsd:sequence>
                <xsd:element ref="conversationNode" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="correlationKeyRefs" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

12. Choreography

Note: The content of this chapter is required for BPMN **Choreography Modeling Conformance** or for BPMN **Complete Conformance**. However, this chapter is not required for BPMN **Process Modeling Conformance**, BPMN **Process Execution Conformance**, or BPMN **BPEL Process Execution Conformance**. For more information about BPMN conformance types, see page 28.

A Choreography is a type of process, but differs in purpose and behavior from a standard BPMN Process. A standard Process, or an *Orchestration* Process (see page 153), is more familiar to most process modelers and defines the flow of Activities of a specific PartnerEntity or organization. In contrast, Choreography formalizes the way business *Participants* coordinate their interactions. The focus is not on orchestrations of the work performed *within* these *Participants*, but rather on the exchange of information (Messages) *between* these *Participants*.

Another way to look at Choreography is to view it as a type of business contract between two (2) or more organizations.

This entails Message (document) exchanges in an orderly fashion: e.g. first a retailer sends a purchase order request to a supplier; next the supplier either confirms or rejects intention to investigate the order; then supplier proceeds to investigate stock for line-items and seeks outside suppliers if necessary; accordingly the supplier sends a confirmation or rejection back; during this period the retailer can send requests to vary the order, etc.

Message exchanges between partners go beyond simple request-response interactions into multi-cast, contingent requests, competing receives, streaming and other service interaction patterns (REF for SIP). Moreover, they cluster around distinct scenarios such as: creation of sales orders; assignment of carriers of shipments involving different sales orders; managing the "red tape" of crossing customs and quarantine; processing payment and investigating exceptions. A Choreography is a definition of expected behavior, basically a procedural business contract, between interacting *Participants* (see page 124 for more information on *Participants*). It brings Message exchanges and their logical relation as Conversations into view. This allows partners to plan their Business Processes for inter-operation without introducing conflicts. An example of a conflict could arise if a retailer was allowed to send a variation on a purchase order immediately after sending the initial request. The Message exchange sequences in Choreography models need to be reflected in the orchestration Processes of participants. A Choreography model makes it possible to derive the Process interfaces of each partner's Process (REF: Decker & Weske, 2007).

To leverage the familiarity of flow charting types of Process models, BPMN Choreographies also have "activities" that are ordered by Sequence Flow. These "activities" consist of one (1) or more *interactions* between *Participants*. These *interactions* are often described as being *message exchange patterns* (MEPs). A MEP is the atomic unit ("Activity") of a Choreography.

Some MEPs involve a single Message (e.g., a "Customer" requests an "Order" from a "Supplier"). Other MEPs will involve two (2) Messages in a request and response format (e.g., a "Supplier" request a "Credit Rating" from a "Financial Institution," who then returns the "Credit Rating" to the "Supplier"). There can be even more complex MEPs that involve error Messages, for example.

A single MEP is defined as a BPMN Choreography Task (see page 350). Thus, a Choreography defines the order in which Choreography Tasks occur. Choreography Sub-Processes allow the composition/decomposition of Choreographies.

Choreographies are designed in BPMN to allow stand-alone, scalable models of these *Participant interactions*. However, since BPMN provides other Business Process modeling views, Choreographies are designed to fit within BPMN Collaboration diagrams to display of the relationship between the Choreography and *Orchestration* Processes (thus, expanding BPMN 1.2 capabilities—see page 143, above, for more information on Collaborations, and page 394 for Choreographies within Collaborations).

Figure 12-1 shows displays the metamodel of the key BPMN elements that contribute to Choreography modeling. The sections of this chapter will describe the characteristics of these elements and how they are used in a Choreography.

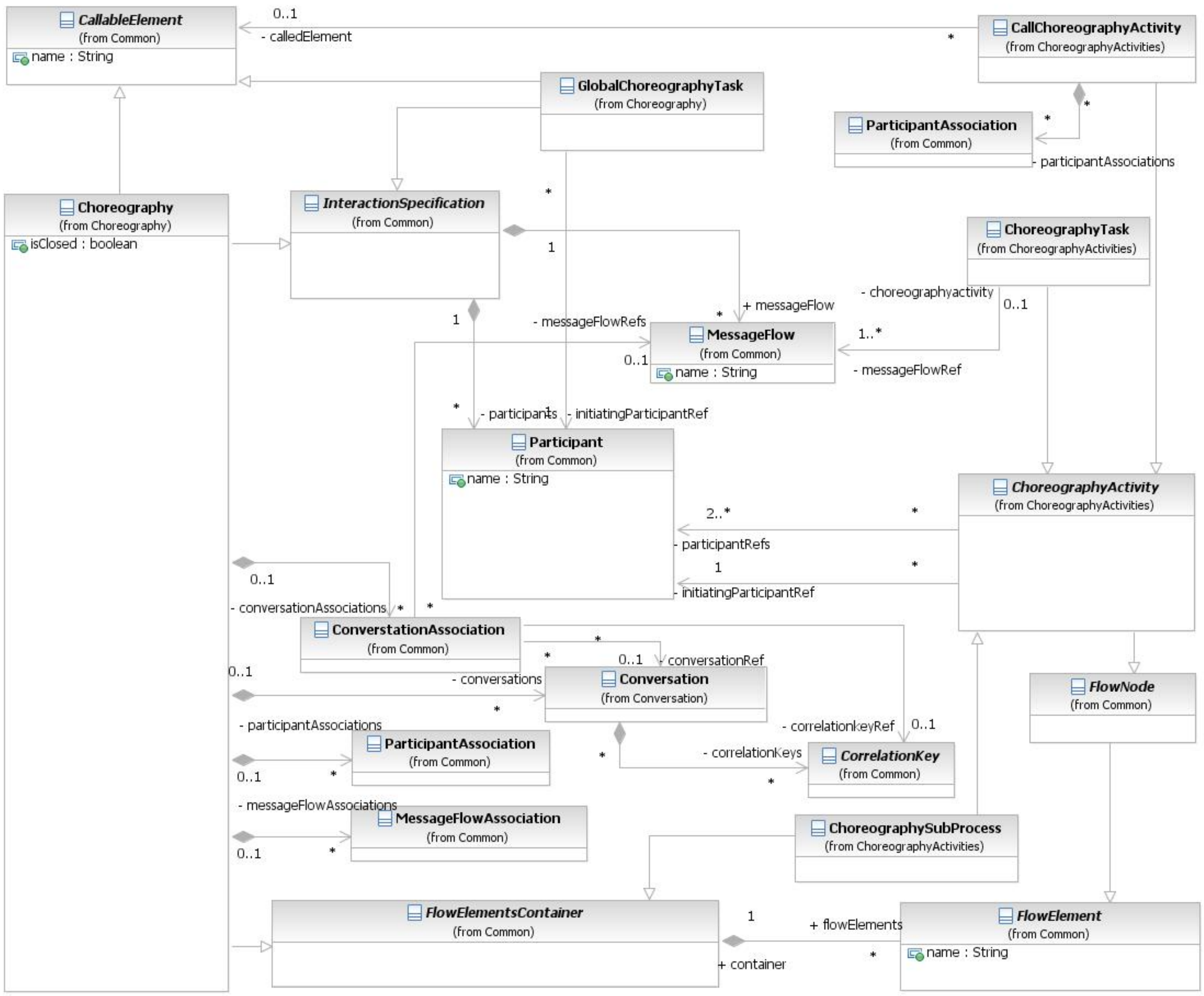


Figure 12-1 – The Choreography metamodel

The Choreography element inherits the attributes and model associations of CallableElement (see Table 8-30) and of FlowElementContainer (see Table 8-46). Table 12-1 presents the additional model associations of the Choreography element.

Table 12-1 – Choreography Model Associations

Attribute Name	Description/Usage
IsClosed: boolean = false	A Boolean value specifying whether Choreography Activities not modeled in the Choreography can occur when the Choreography is carried out. If the value is *true*, they MAY NOT occur. If the value is *false*, they MAY occur.
conversations: Conversation [0..*]	The conversation model aggregation relationship allows to have Conversations contained in a Choreography, to group Message Flow of the Choreography and associate *correlation* information. Such a Conversation SHOULD only use Message Flow references to group the Message Flow of the enclosing Choreography.
conversationAssociations: ConversationAssociation [0..*]	This attribute provides the list of ConversationAssociations that are used to apply a Conversation to the Choreography. The ConversationAssociations is used to identify the Message Flow that are grouped by the referenced Conversation. This grouping can be done automatically through the CorrelationKey of the Conversation (matching the CorrelationKey to the Messages of the Message Flow) or done through user selection if a CorrelationKey has not been defined. If the Conversation lists *Participants*, then the participantAssociations (see below) are used to map the *Participants* of the Conversation to the *Participants* of the Collaboration. If the Conversation lists Message Flow, then the conversationMessageFlowAssociations (see below) are used to map the Message Flow of the Conversation to the Message Flow of the Collaboration.
participantAssociations: Participant Association [0..*]	This attribute provides a list of mappings from the *Participants* of a referenced Choreography or Conversation to the *Participants* of the Choreography.
messageFlowAssociations: Message Flow Association [0..*]	This attribute provides a list of mappings from the Message Flow of a referenced Choreography or Conversation to the Message Flow of the Choreography.

12.1. Basic Choreography Concepts

A key to understanding Choreographies and how they are used in BPMN is their relationship to Pools (see page 124 for more information on Pools). Choreographies exist outside of or in between Pools. A Process, within a Pool, represents the work of a specific PartnerEntity (e.g., “FedEx”), often substituted by a PartnerRole (e.g., “Shipper”) when a PartnerEntity is not identified and can be decided later. The PartnerEntity/PartnerRole is called a *Participant* in BPMN. Pools are the graphical representation of *Participants*. A Choreography, on the other hand, is a different kind of process. A Choreography defines the sequence of *interactions* between *Participants*. Thus, a Choreography does not exist in a single Pool—it is not the purview of a single *Participant*. Each step in the Choreography involves two (2) or more

Participants (these steps are called Choreography Activities—see below). This means that the Choreography, in BPMN terms, is defined outside of any particular Pool.

The key question that needs to be continually asked during the development of a Choreography is "what information do the *Participants* in the Choreography have?" Basically, each *Participant* can only understand the status of the Choreography through observable behavior of the other *Participants*–which are the Messages that have been sent and received. If there are only two (2) *Participants* in the Choreography, then it is very simple—both *Participants* will be aware of who is responsible for sending the next Message. However, if there are more than two (2) *Participants*, then the modeler must be careful to sequence the Choreography Activities in such a way that the *Participants* know when they are responsible for initiating the *interactions*.

Figure 12-2 presents a sample Choreography. The details of Choreography behavior and elements will be described in the sections below.

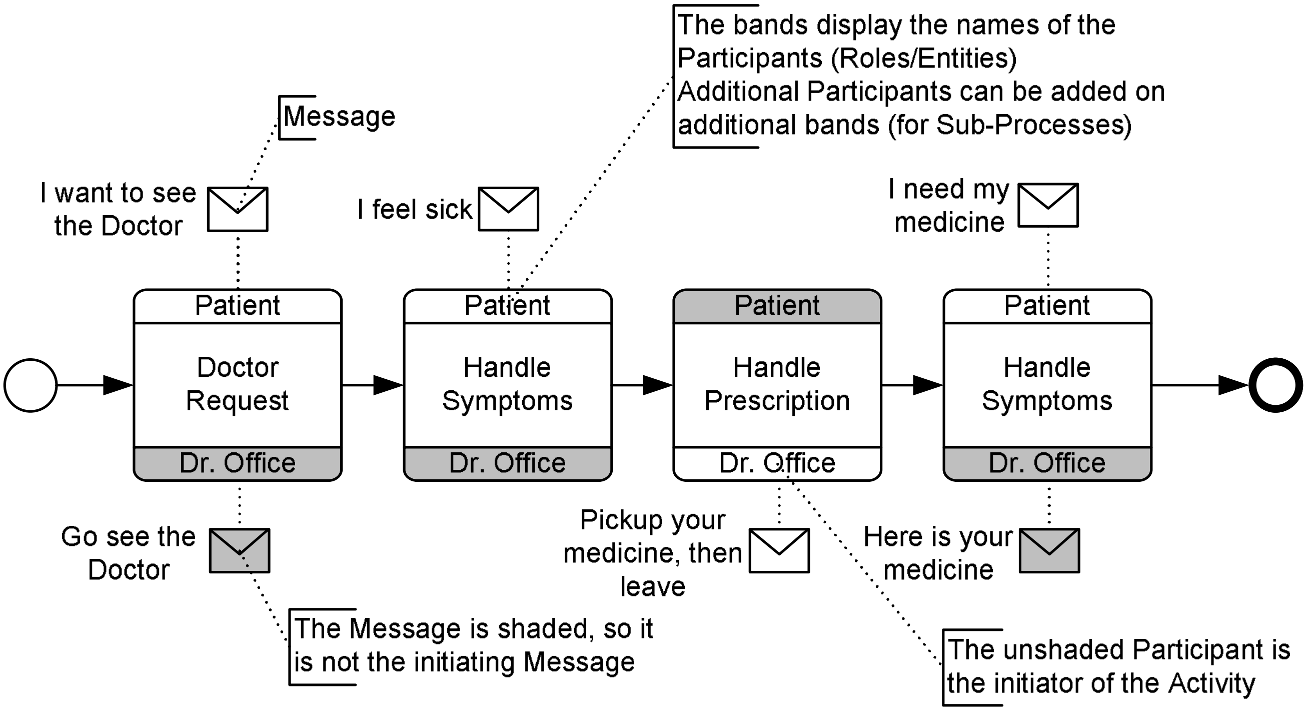

The bands display the names of the Participants (Roles/Entities)
Additional Participants can be added on additional bands (for Sub-Processes)
Message
I want to see the Doctor
I feel sick
I need my medicine
Patient
Doctor Request
Dr. Office
Patient
Handle Symptoms
Dr. Office
Patient
Handle Prescription
Dr. Office
Patient
Handle Symptoms
Dr. Office
Go see the Doctor
Pickup your medicine, then leave
Here is your medicine
The Message is shaded, so it is not the initiating Message
The unshaded Participant is the initiator of the Activity

Figure 12-2 – An example of a Choreography

To illustrate the correspondence between Collaboration and Choreography, consider an example from logistics. Figure 12-3 shows a Collaboration where the Pools are expanded to reveal orchestration details per participant (for *Shipper, Retailer* etc). Message Flow connect the elements in the different Pools related to different participants, indicating Message exchanges. For example, a *Planned Order Variations* Message is sent by the *Supplier* to the *Retailer*; the corresponding send and receive have been modeled using regular BPMN messaging Events. Also, a number of Messages of the same type being sent, for example a number of *Retailer Order and Delivery Variations* Messages can be sent from the *Retailer* to the *Supplier*, indicated by respective *multi-instances* constructs (for brevity, the actual elements for sending/receiving inside the *multi-instances* construct have been omitted).

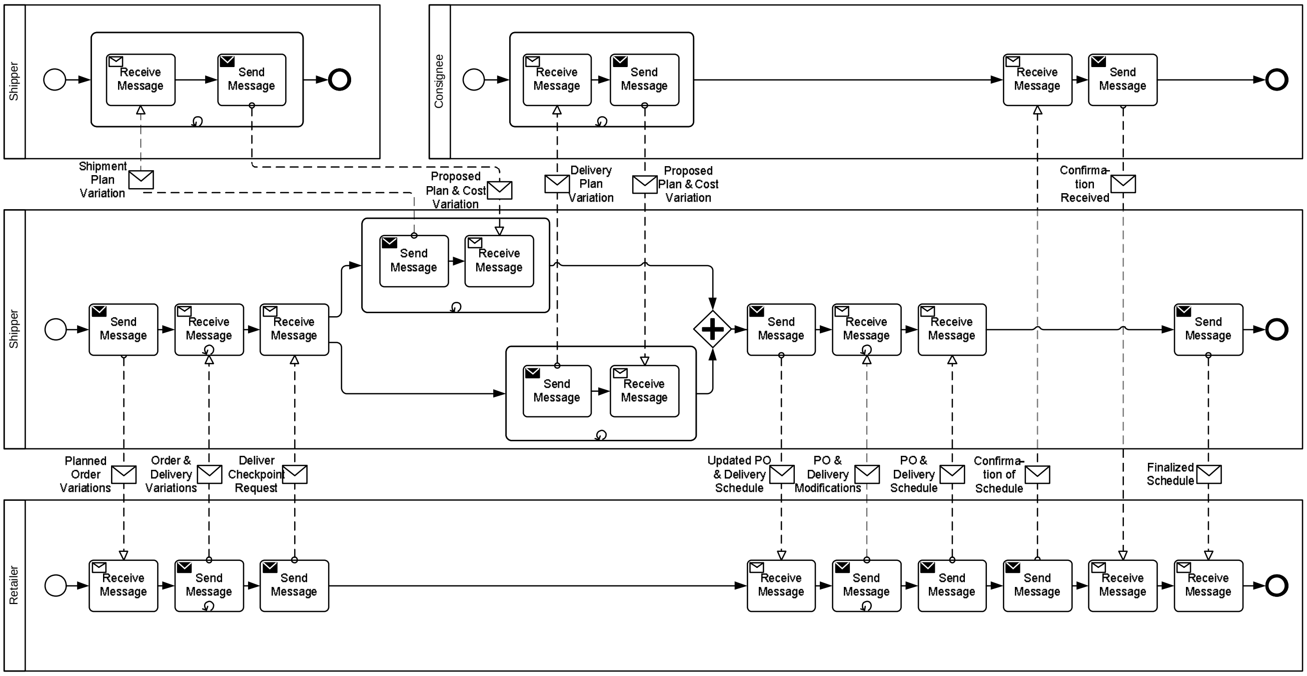


Figure 12-3 – A Collaboration diagram logistics example

The scenario modeled in Figure 12-4 entails shipment planning for the next supply replenishment variations: the *Supplier* confirms all previously accepted variations for delivery with the *Retailer*; the *Retailer* sends back a number of further possible variations; the *Supplier* requests to the *Shipper* and *Consignee* possible changes in delivery; accordingly, the *Retailer* interacts with the *Supplier* and *Consignee* for final confirmations.

A problem with model interconnections for complex Choreographies is that they are vulnerable to errors – interconnections may not be sequenced correctly, since the logic of Message exchanges is considered from each partner at a time. This in turn leads to deadlocks. For example, consider the PartnerRole of *Retailer* in Figure 12-4 and assume that, by error, the order of *Confirmation Delivery Schedule* and *Retailer Confirmation received* (far right) were swapped. This would result in a deadlock since both, *Retailer* and *Consignee* would wait for the other to send a Message. Deadlocks in general, however, are not that obvious and might be difficult to recognize in a Collaboration.

Figure 12-4 shows the Choreography corresponding to the Collaboration of Figure 12-3 above.

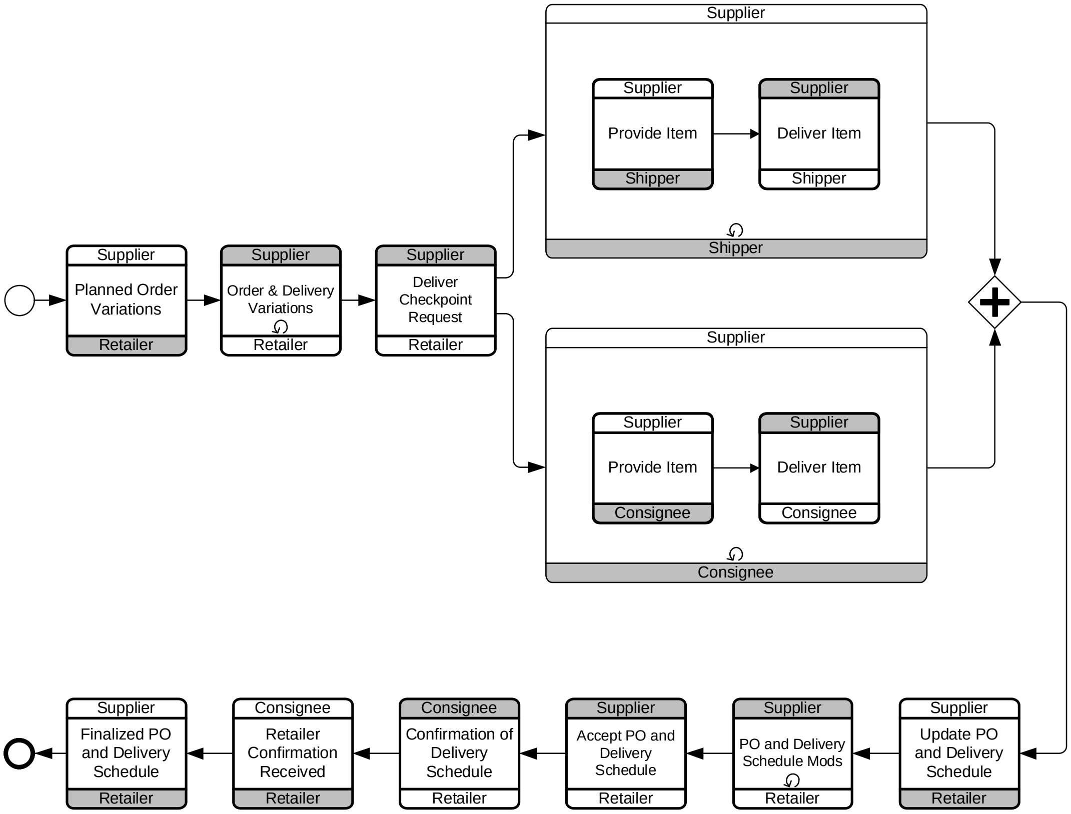


Figure 12-4 – The corresponding Choreography diagram logistics example

12.2. Data

A Choreography does not have a central control mechanism and, thus, there is no mechanism for maintaining any central Process (Choreography) data. Thus, any element in a Process that would normally depend on conditional or assignment expressions, would not have any central source for this data to be maintained and understood by all the *Participants* involved in the Choreography.

As mentioned above, neither Data Objects nor Repositories are used in Choreographies. Both of these elements are used exclusively in Processes and require the concept of a central locus of control. Data Objects are basically variables and there would be no central system to manage them. *Data* can be used in *expressions* that are used in Exclusive Gateways, but only that data which has been sent through a Message in the Choreography.

12.3. Use of BPMN Common Elements

Some BPMN elements are common to both Process and Choreography diagrams, as well as Collaboration; they are used in these diagrams. The next few sections will describe the use of Messages, Message Flow, *Participants*, Sequence Flow, Artifacts, *Correlations*, *Expressions*, and *Services* in Choreography.

The key graphical elements of Gateways and Events are also common to both Choreography and Process. Since their usage has a large impact, they are described in major sections of this chapter (see page 369 for Events and page 375 for Gateways).

12.3.1. Sequence Flow

Sequence Flow are used within Choreographies to show the sequence of the Choreography Activities, which may have intervening Gateways. They are used in the same way as they are in Processes. They are only allowed to connect with other *Flow Objects*. For Processes, they can only connect Events, Gateways, and Activities. For Choreographies, they can only connect Events, Gateways, and Choreography Activities (see Figure 12-5).

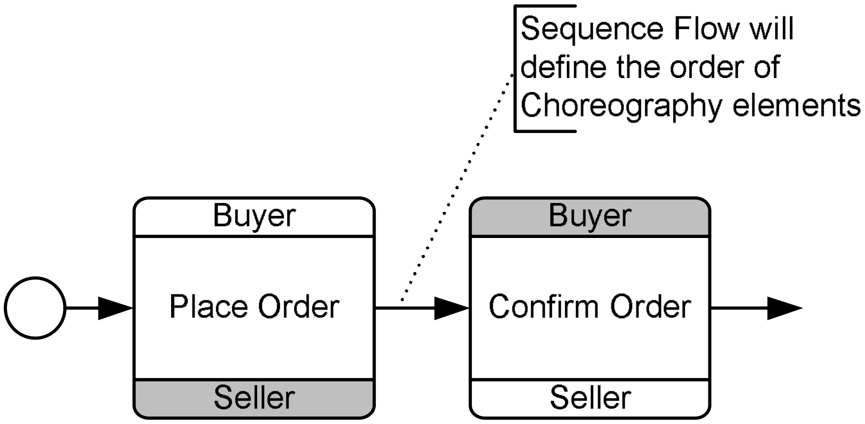

Sequence Flow will define the order of Choreography elements
Buyer
Place Order
Seller
Buyer
Confirm Order
Seller

Figure 12-5 – The use of Sequence Flow in a Choreography

There are two additional variations of Sequence Flow:

- Conditional Sequence Flow: *Conditions* can be added to Sequence Flow in two situations:
  - From Gateways: *Outgoing* Sequence Flow have *conditions* for Exclusive and Inclusive Gateways. The data referenced in the *conditions* must be visible to two (2) or more *Participants* in the Choreography. The data becomes visible if it is part of a Message that had been sent (previously) within the Choreography. See pages 375 and 383 for more information about how Exclusive and Inclusive Gateways are used in Choreography.
  - From Choreography Activities: *Outgoing* Sequence Flow may have *conditions* for Choreography Activities. Since these act similar to Inclusive Gateways, the Conditional Sequence Flow can be used in Choreographies. The *conditions* have the same restrictions that apply to the visibility of the data for Gateways.
- Default Sequence Flow: For Exclusive Gateways, Inclusive Gateways, and Choreography Activities that have *Conditional* Sequence Flow, one of the *outgoing* Sequence Flow may be a *Default* Sequence Flow. Because the other *outgoing* Sequence Flow will have appropriately visible of data as described above, the *Participants* would know if all the other *conditions* would be *false*, thus the *Default* Sequence Flow would be selected and the Choreography would move down that Sequence Flow.

In some applications it is useful to allow more Messages to be sent between *Participants* when a Choreography is carried out than are contained the Choreography model. This enables *Participants* to exchange other Messages as needed without changing the Choreography. There are two ways to specify this:

- If the isClosed attribute of a Choreography has a value of *false* or no value, then *Participants* MAY send Messages to each other without additional Choreography Activities in the Choreography. Unmodeled messaging can be restricted on particular Sequence Flow in the Choreography, see next bullet. If the isClosed attribute of a Choreography has a value of *true*, then *Participants* MAY NOT send Messages to each other without additional Choreography Activities in the Choreography. This restriction overrides any unmodeled messaging allowed by Sequence Flow in the next bullet.
- If the isImmediate attribute of a Sequence Flow has a value of *false* or no value, then *Participants* MAY send Messages to each other between the elements connected by the Sequence Flow without additional Choreography Activities in the Choreography. If the isImmediate attribute of a Sequence Flow has a value of *true*, then *Participants* MAY NOT send Messages to each other between the elements connected by the Sequence Flow without additional Choreography Activities in the Choreography. The value of isImmediate attribute of a Sequence Flow has no effect if the isClosed attribute of the containing Choreography has a value of *true*.

Restrictions on unmodeled messaging specified with isClosed and isImmediate applies only under the Choreography containing the restriction. PartnerEntities and PartnerRoles of the *Participants* MAY send Messages to each other under other Choreographies, Collaborations, and Conversations.

12.3.2. Artifacts

Both Text Annotations and Groups can be used within Choreographies and all BPMN diagrams. There are no restrictions on their use.

12.3.3. Correlations

Correlation will only indirectly impact a Choreography.

12.4. Choreography Activities

A Choreography Activity represents a point in a Choreography flow where an *interaction* occurs between two (2) or more *Participants*.

The Choreography Activity class is an abstract element, sub-classing from FlowElement (as shown in Figure 12-6). When Choreography Activities are defined they are contained within a Choreography or a Choreography Sub-Process, which are FlowElementContainers (other FlowElementContainers are not allowed to contain Choreography Activities).

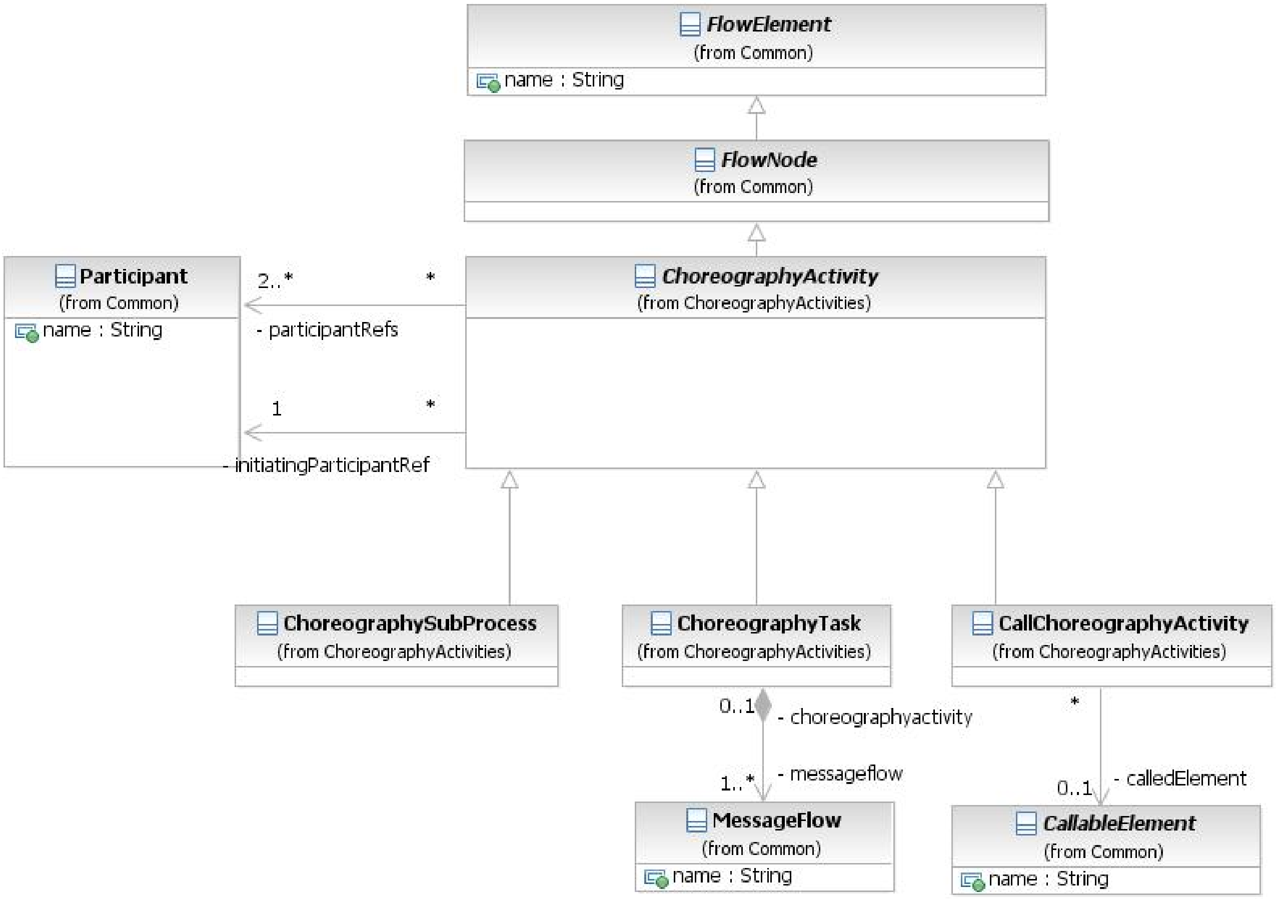


Figure 12-6 – The metamodel segment for a Choreography Activity

The Choreography Activity element inherits the attributes and model associations of FlowElement (see Table 8-45) through its relationship to FlowNode. Table 12-2 presents the additional model associations of the Choreography Activity element

Table 12-2 – Choreography Activity Model Associations

Attribute Name	Description/Usage
participantRefs: Participant [2..*]	A Choreography Activity has two (2) or more *Participants* (see page 146 for more information on *Participants*).
initiatingParticipant: Participant	One (1) of the *Participants* will be the one that initiates the Choreography Activity.

12.4.1. Choreography Task

A Choreography Task is an atomic Activity in a Choreography Process. It represents an *Interaction*, which is a coherent set (1 or more) of Message exchanges between two (2) *Participants*. Using a Collaboration diagram to view these elements (see page 143 for more information on Collaboration), we would see the two (2) Pools representing the two (2) *Participants* of the *Interaction* (see Figure 12-7). The communication between the *Participants* is shown as a Message Flow.

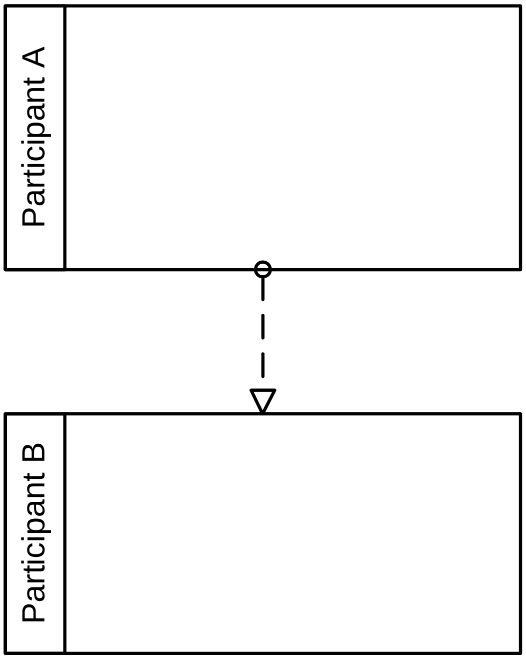


Figure 12-7 – A Collaboration view of Choreography Task elements

In a Choreography diagram, this *Interaction* is collapsed into a single object, a Choreography Task. The name of the Choreography Task and each of the *Participants* are all displayed in the different bands that make up the shape's graphical notation. There are two (2) more Participant Bands and one Task Name Band (see Figure 12-8).

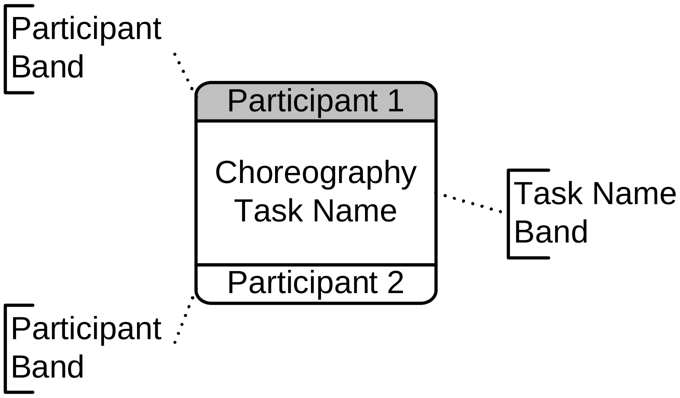


Figure 12-8 – A Choreography Task

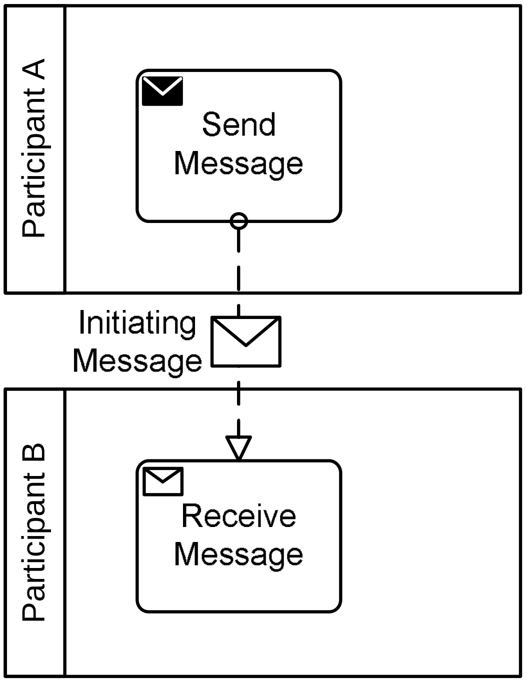


Figure 12-9 – A Choreography Task

The interaction defined by a Choreography Task can be shown in an expanded format through a Collaboration diagram (see Figure 12-7—see page 143 for more information on Collaborations). In the Collaboration view, the *Participants* of the Choreography Task Participant Band's will be represented by Pools. The interaction between them will be a Message Flow.

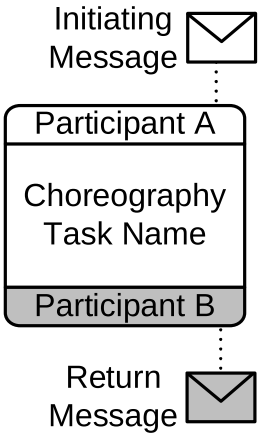


Figure 12-10 – A two-way Choreography Task

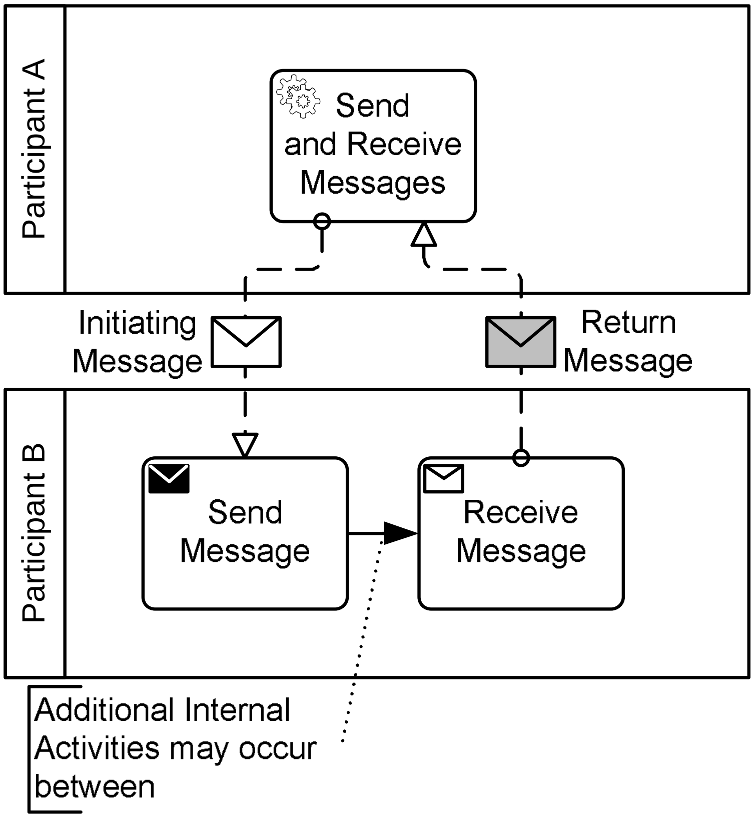


Figure 12-11 – A Choreography Task

In a Choreography Diagram, the Choreography Task object shares the same shape as a Task or any other BPMN Activity, which is a rectangle that has rounded corners.

- A Choreography Task is a rounded corner rectangle that MUST be drawn with a single line.
  - The use of text, color, size, and lines for a Choreography Task MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63.

The three (3) bands in the Choreography Task shape provide the distinction between this type of Task and an Orchestration Task (in a traditional BPMN diagram).

As with a standard Orchestration Task, the Choreography Task may have internal markers to show how the Choreography Task may be repeated. There are two types of internal markers (see Figure 12-12):

- A Choreography Task MAY have only one of the two (2) markers at one time.
  - The marker for a Choreography Task that is a standard *loop* MUST be a small line with an arrowhead that curls back upon itself.
  - The marker for a Choreography Task that is *multi-instance* MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the Task Name Band of the shape.

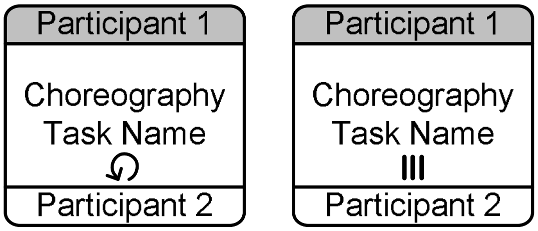


Figure 12-12 – Choreography Task Markers

Figure 12-13

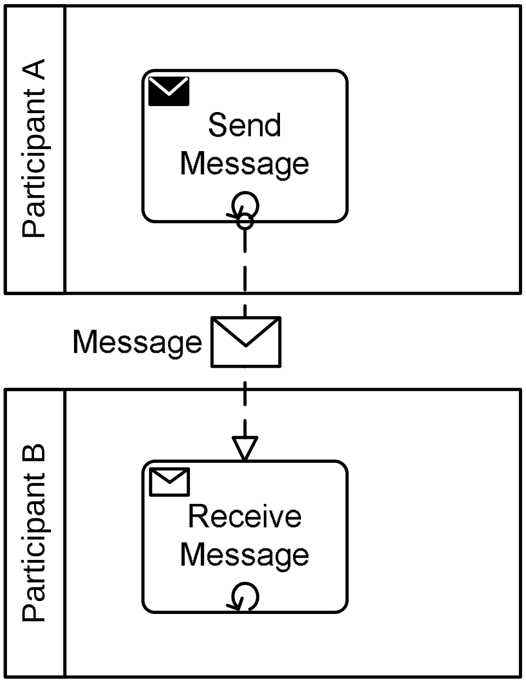


Figure 12-13 – The Collaboration view of a *looping* Choreography Task

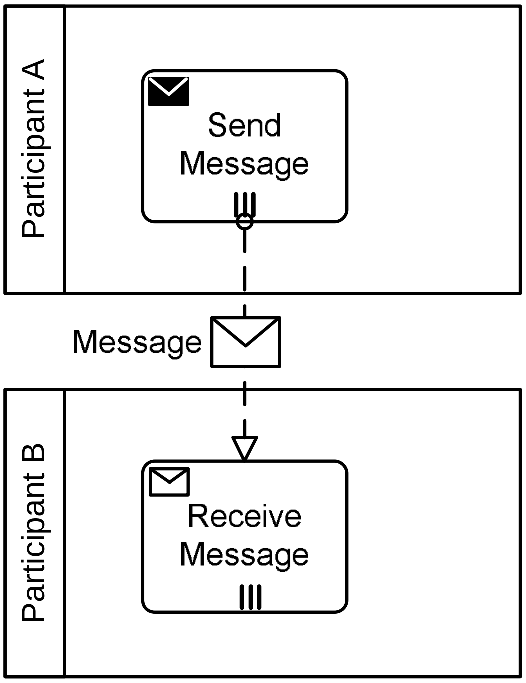


Figure 12-14 – The Collaboration view of a *Multi-Instance* Choreography Task

There are situations when a *Participant* for a `Choreography Task` is actually a *multi-instance Participant*. A *multi-instance Participant* represents a situation where there are more than one possible related *Participants* (`PartnerRoles`/`PartnerEntities`) that may be involved in the `Choreography`. For example, in a `Choreography` that involves the shipping of a product, there may be more than one type of shipper used, depending on the destination. When a *Participant* in a `Choreography` contains multiple *instances*, then a *multi-instance* marker will be added to the `Participant Band` for that *Participant* (see Figure 12-15).

- The marker for a `Choreography Task` that is *multi-instance* MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the `Participant Band` of the shape.
    - The width of the `Participant Band` will be expanded to contain both the name of the *Participant* and the *multi-instance* marker.

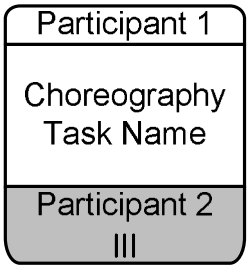


Figure 12-15 – A Choreography Task with a multiple *Participant*

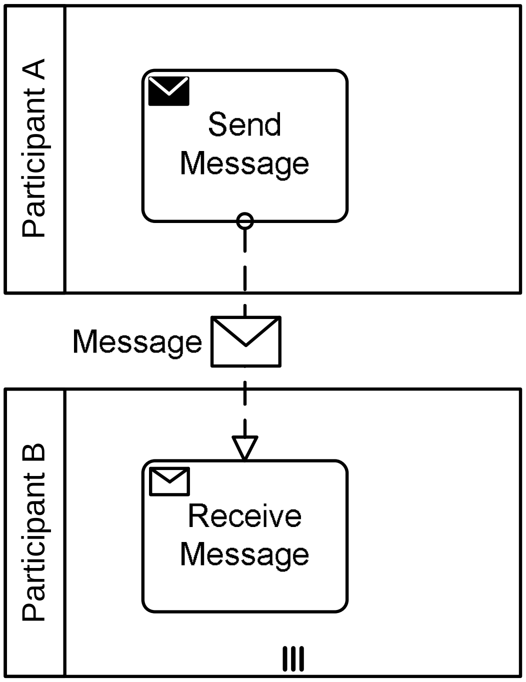


Figure 12-16 – A Collaboration view of a Choreography Task with a multiple *Participant*

The Choreography Task element inherits the attributes and model associations of Choreography Activity (see Table 12-2). Table 12-3 presents the additional model associations of the Choreography Task element.

Table 12-3 – Choreography Task Model Associations

Attribute Name	Description/Usage
messageFlow: Message Flow [1..*]	Although not graphical represented, Choreography Task contain one (1) or more Message Flow that represent the interaction(s) between the *Participants* referenced by the Choreography Task.

12.4.2. Choreography Sub-Process

A Choreography Sub-Process is a compound Activity in that it has detail that is defined as a flow of other Activities, in this case, a Choreography. Each Choreography Sub-Process involves two (2) or more *Participants*. The name of the Choreography Sub-Process and each of the *Participants* are all displayed in the different bands that make up the shape's graphical notation. There are two (2) more Participant Bands and one Sub-Process Name Band.

The Choreography Sub-Process can be in a collapsed view that hides its details (see Figure 12-17) or a Choreography Sub-Process can be expanded to show its details (a Choreography Process) within the Choreography Process in which it is contained (see Figure 12-19). In the collapsed form, the Sub-Process object uses a marker to distinguish it as a Choreography Sub-Process, rather than a Choreography Task.

- The Sub-Process marker MUST be a small square with a plus sign (+) inside. The square MUST be positioned at the bottom center of the Sub-Process Name Band within the shape.

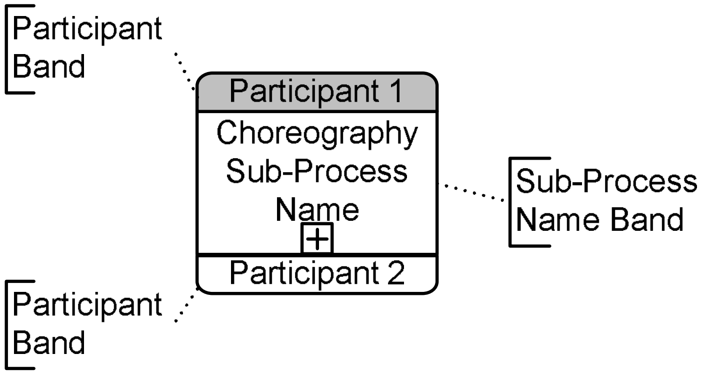


Figure 12-17 – A Choreography Sub-Process

Figure 12-18 shows an example of a potential Collaboration view of the above Choreography Sub-Process.

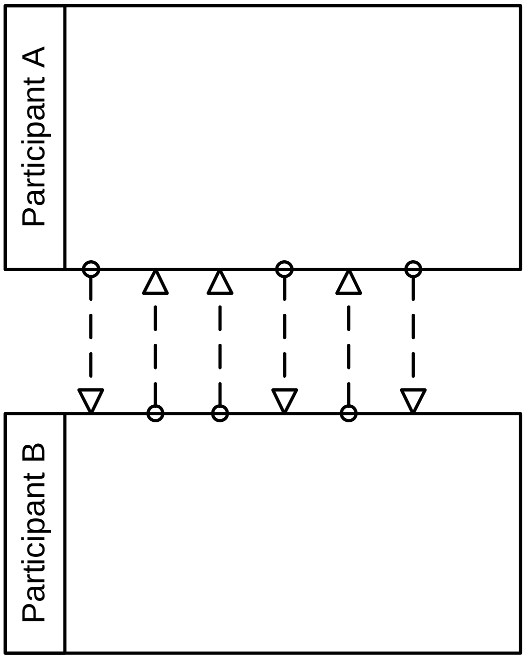


Figure 12-18 – A Collaboration view of a Choreography Sub-Process

Figure 12-19 shows an example of an *expanded* Choreography Sub-Process.

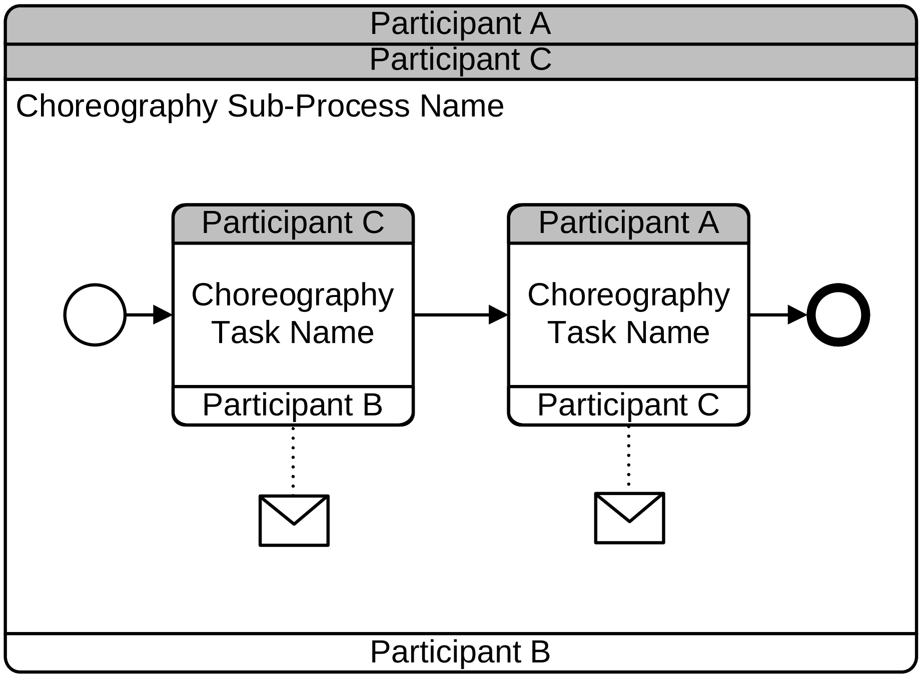


Figure 12-19 – An expanded Choreography Sub-Process

Figure 12-20 shows an example of a potential Collaboration view of the above Choreography Sub-Process.

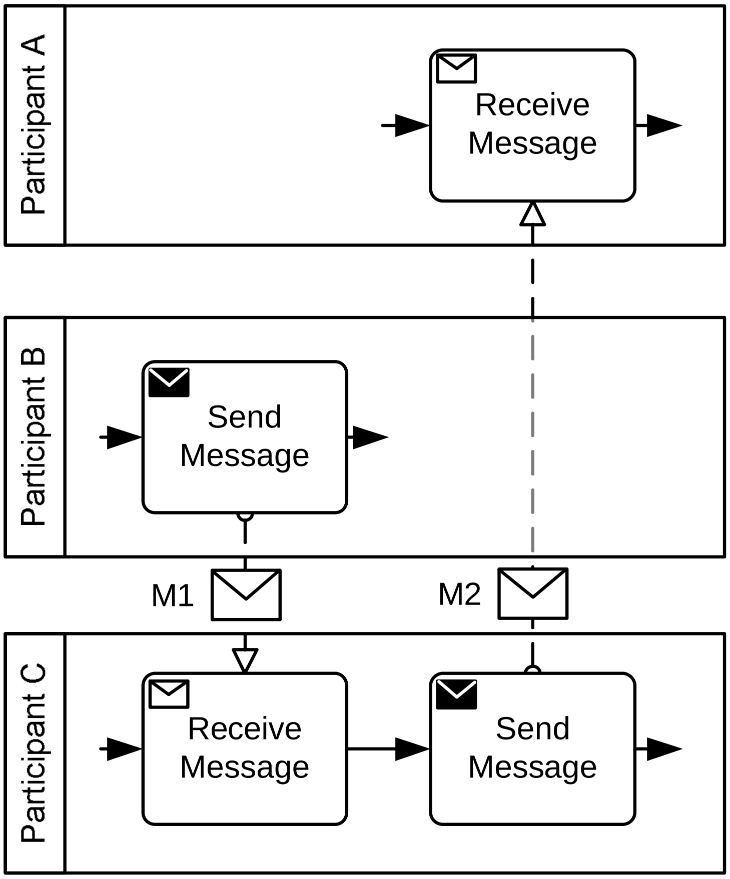


Figure 12-20 – A Collaboration view of an expanded Choreography Sub-Process

The *Parent* Choreography Sub-Process (Expanded)

The Choreography Activity shares the same shape as a Sub-Process or any other BPMN Activity, which is in this state.

- A Choreography Sub-Process is a rounded corner rectangle that MUST be drawn with a single thin line.
  - The use of text, color, size, and lines for a Choreography Sub-Process MUST follow the rules defined in Section “Use of Text, Color, Size, and Lines in a Diagram” on page 63.

The three (3) or more partitions in the Choreography Sub-Process shape provide the distinction between this type of Task and an Orchestration Sub-Process (in a traditional BPMN diagram).

It is possible for a Choreography Sub-Process to involve more than two (2) *Participants*. In this case, an additional Participant Band will be added to the shape for each additional *Participant* (see Figure 12-21). The ordering and position of the Participant Band (either in the upper or lower positions) is up to the modeler or modeling tool. In addition, any Participant Band beyond the first two optional; it is displayed at the discretion of the modeler or modeling tool. However, each Participant Band that is added MUST be added to the upper and lower sections of the Choreography Sub-Process in an alternative manner.

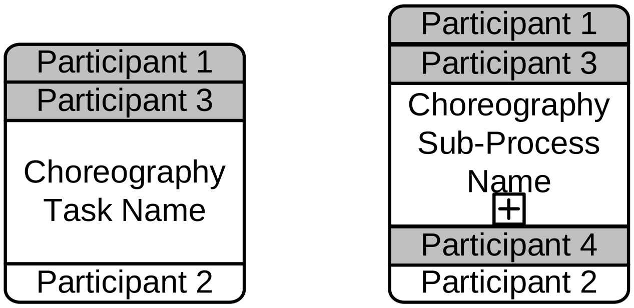


Figure 12-21 –Choreography Sub-Process (Collapsed) with More than Two (2) *Participants*

As with a standard Orchestration Sub-Process, the Choreography Sub-Process may have internal markers to show how the Choreography Sub-Process may be repeated. There are two types of internal markers (see Figure 12-22):

- A Choreography Sub-Process MAY have only one of the two (2) markers at one time.
  - The marker for a Choreography Sub-Process that is a standard *loop* MUST be a small line with an arrowhead that curls back upon itself.
  - The marker for a Choreography Sub-Process that is multi-*instance* MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the Sub-Process Name Band of the shape.

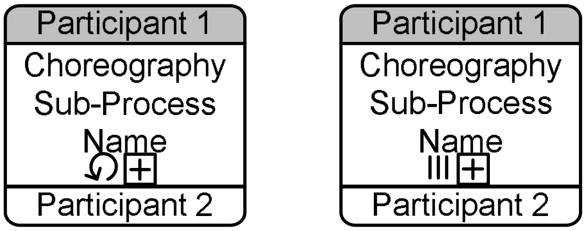


Figure 12-22 – Choreography Sub-Process Markers

There are situations when a *Participant* for a Choreography Sub-Process is actually a *multi-instance Participant*. A *multi-instance Participant* represents a situation where there are more than one possible related *Participants* (PartnerRoles/PartnerEntities) that may be involved in the Choreography. For example, in a Choreography that involves the shipping of a product, there may be more than one type of shipper used, depending on the destination. When a *Participant* in a Choreography contains multiple *instances*, then a *multi-instance* marker will be added to the Participant Band for that *Participant* (see Figure 12-23).

- The marker for a Choreography Sub-Process that is multi-*instance* MUST be a set of three vertical lines.
- The marker that is present MUST be centered at the bottom of the Participant Band of the shape.
  - The width of the Participant Band will be expanded to contain both the name of the *Participant* and the *multi-instance* marker.

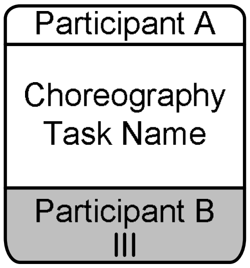


Figure 12-23 – Choreography Sub-Process Markers

This includes Compensation Event Sub-Processes (contained within a Choreography Sub-Process) as well as the external Compensation Activity connected through an Association.

The Choreography Sub-Process element inherits the attributes and model associations of Choreography Activity (see Table 12-2). The Choreography Sub-Process does not have any additional attributes or model associations.

12.4.3. Call Choreography Activity

A Call Choreography Activity identifies a point in the Process where a global Choreography or a Global Choreography Task is used. The Call Choreography Activity acts as a place holder for the inclusion of the Choreography element it is calling. This pre-defined called Choreography element becomes a part of the definition of the *parent* Choreography.

A Call Choreography Activity object shares the same shape as the Choreography Task and Choreography Sub-Process, which is a rectangle that has rounded corners, two (2) or more Participant Bands, and an Activity Name Band. However, the target of what the Choreography Activity calls will determine the details of its shape.

- If the Call Choreography Activity calls a Global Choreography Task, then the shape will be the same as a Choreography Task, but the boundary of the shape will MUST have a thick line (see Figure 12-24)
- If the Call Choreography Activity calls a Choreography, then there are two (2) options:
    - The details of the called Choreography can be hidden and the shape will be the same as a *collpased* Choreography Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 12-25).
    - The details of the called Choreography can be shown and the shape will be the same as an *expanded* Choreography Sub-Process, but the boundary of the shape MUST have a thick line (see Figure 12-26).

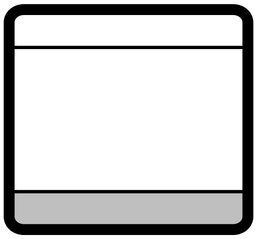


Figure 12-24 – A Call Choreography Activity calling a Global Choreography Task

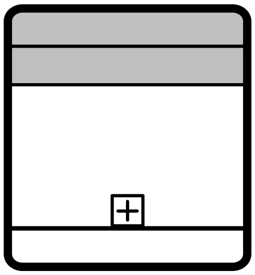


Figure 12-25 – A Call Choreography Activity calling a Choreography (Collapsed)

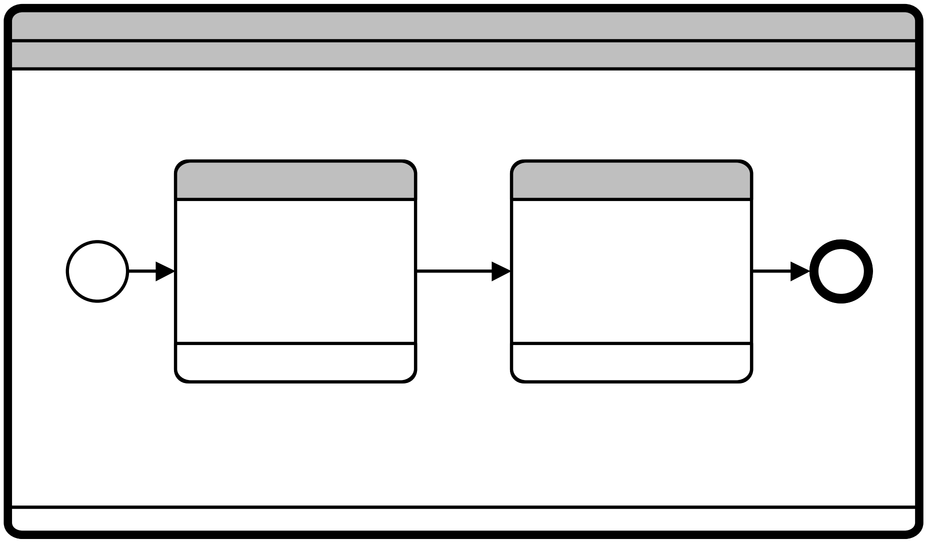


Figure 12-26 – A Call Choreography Activity calling a Choreography (expanded)

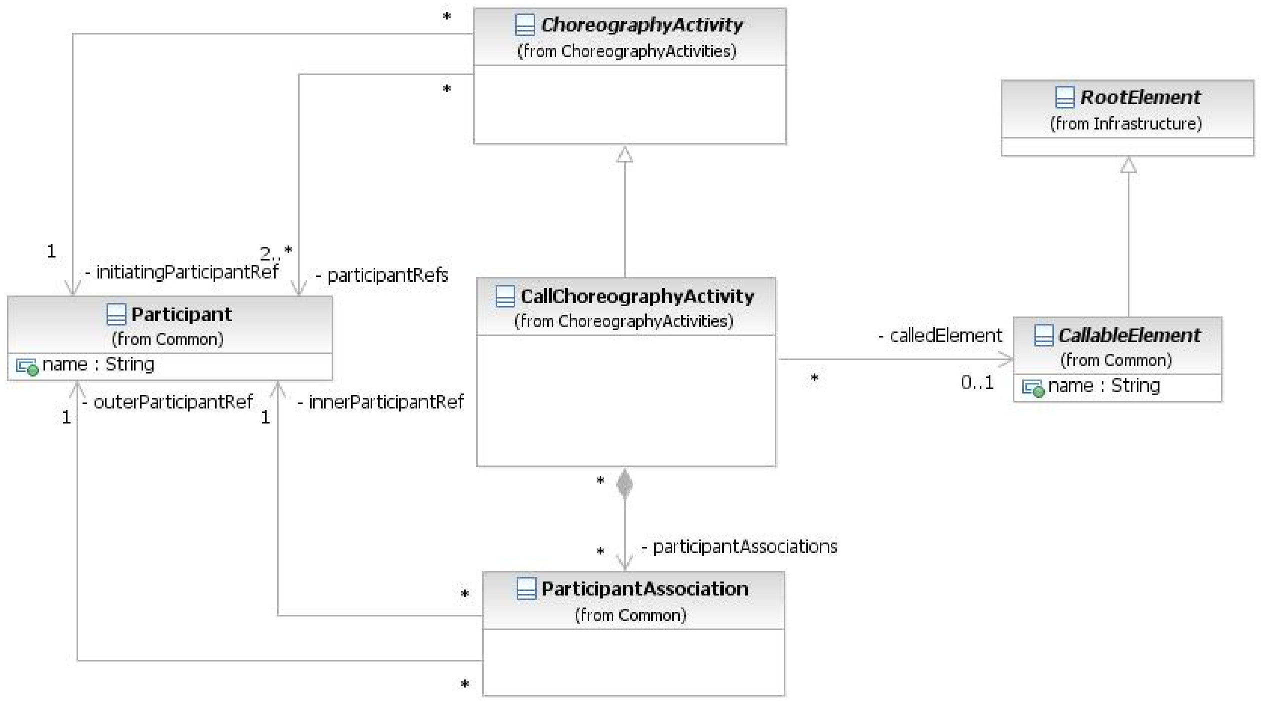


Figure 12-27 – The Call Choreography Activity class diagram

The Call Choreography Activity element inherits the attributes and model associations of ChoreographyActivity (see Figure 12-27 and Table 12-2). Table 12-4 presents the additional model associations of the Call Choreography Activity element

Table 12-4 – Call Choreography Activity Model Associations

Attribute Name	Description/Usage
calledElement: CallableElement [0..1]	The element to be called, which will be either a Choreography or a GlobalChoreographyTask. Other CallableElements, such as Process, GlobalTask, Conversation, and GlobalCommunication MUST NOT be called by the Call Conversation element.
participantAssociations: ParticipantAssociation [0..*]	Specifies how *Participants* in a nested Choreography or Global Choreography Task match up with the *Participants* in the Choreography containing the Call Choreography Activity.

12.4.4. Global Choreography Task

A GlobalChoreographyTask is a reusable, atomic Choreography Task definition that can be called from within any Choreography by a Call Choreography Activity.

The GlobalChoreographyTask element inherits the attributes and model associations of CallableElement (see Table 8-30) and InteractionSpecification (see Table 8-48). Table 12-5 presents the additional model associations of the GlobalChoreographyTask element

Table 12-5 – Global Choreography Task Model Associations

Attribute Name	Description/Usage
initiatingParticipantRef: Participant	One (1) of the *Participants* will be the one that initiates the Global Choreography Task.

12.4.5. Looping Activities

Both Choreography Sub-Processes can have *standard loops* and *multi-instances*. The data used to define the *loop conditions* must be visible to all *Participants*

12.4.6. The Sequencing of Activities

There are constraints on how Choreography Activities can be sequenced (through Sequence Flow) in a Choreography. These constraints are due to the limited visibility of the *Participants*, which only know of the progress of the Choreography by the Messages that occur. When a *Participant* sends or receives a Message, then that *Participant* knows exactly how far the Choreography has progressed. This means that the ordering of Choreography Activities must take into account when the *Participants* send or receive Messages so that they *Participants* are not required to guess about when it is their turn to send a Message.

The basic rule of Choreography Activity sequencing is this:

- The *Initiator* of a Choreography Activity MUST have been involved (as *Initiator* or *Receiver*) in the previous Choreography Activity.

Of course, the first Choreography Activity in a Choreography does not have this constraint.

Figure 12-28 shows a sequence of two (2) Choreography Activities that follow this constraint. “Participant B” is the *Initiator* of “Choreography Task 2” after being the *Receiver* in “Choreography Task 1.” While there is no requirement that “Participant B” sends the Message immediately, since there may be internal work that the *Participant* needs to do prior to the Message. But in this situation there is no ambiguity that “Participant B” will be the *Initiator* of the next Choreography Task. “Participant C” does not know exactly when the Message will arrive from “Participant B,” but “Participant C” knows that one will arrive and there are not any additional requirements on the *Participant* until the Message arrives.

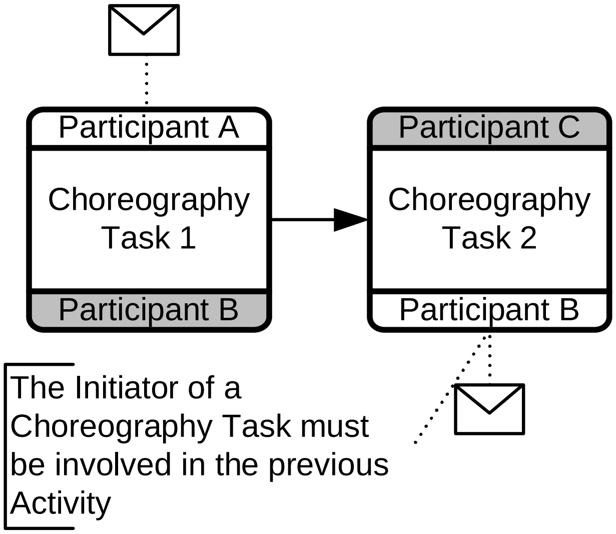


Figure 12-28 – A valid sequence of Choreography Activities

Naturally, the sequence of Choreography Activities shown in Figure 12-28, above can be expanded into a Collaboration diagram to show how the sequence can be enforced. Figure 12-29 shows the corresponding Collaboration. The diagram shows how the Activities within the individual Pools fit with the design of the Choreography.

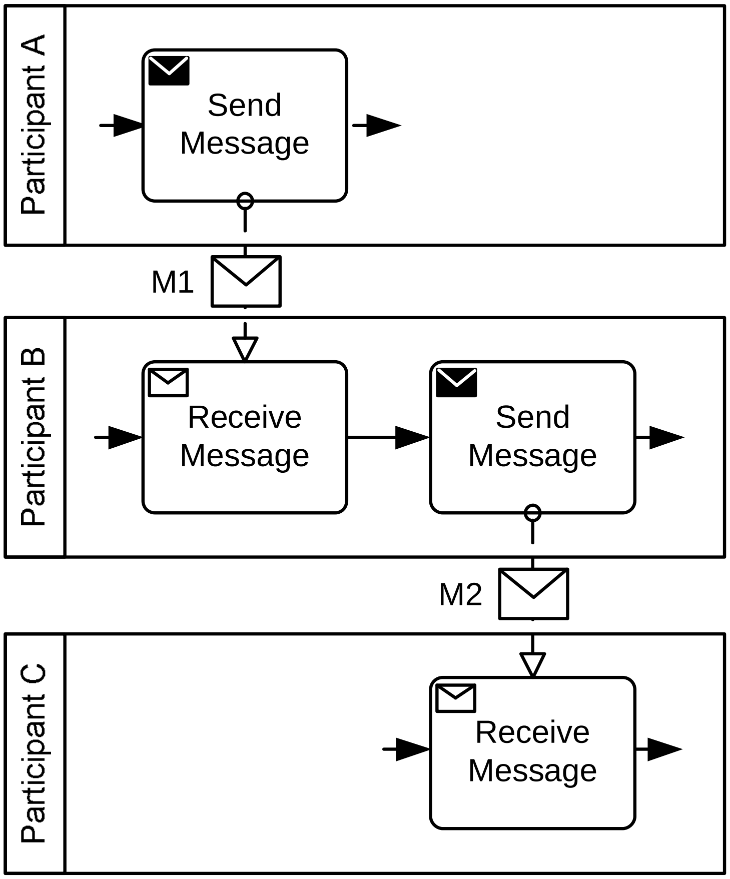


Figure 12-29 – The corresponding Collaboration for a valid Choreography sequence

When determining a valid sequence of Choreography Tasks, it is important to consider the type of Choreography Tasks that are being used. A single Choreography Task can be used for one (1) or more Messages. Most of the time there will be one (1) or two (2) Messages for a Choreography Task. Figure 12-30 shows a sequence of Choreography Tasks, the first one being a two-way interaction, where the initiator sends a Message and gets a response from the other *Participant*.

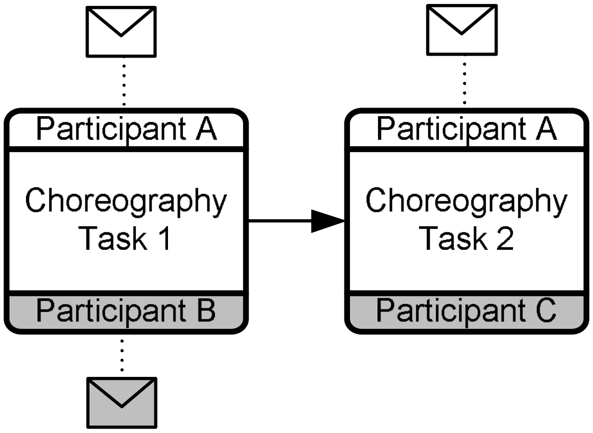


Figure 12-30 – A valid sequence of Choreography Activities with a two-way Activity

Figure 12-31 shows the corresponding Collaboration and how the two Choreography Tasks are reflected in the Processes within the Pools. The Choreography Task that has two Messages will is reflected by three Process Tasks. Usually in these cases, the initiating Participant will use a single Activity to handle both the sending and receiving of the Messages. A BPMN Service Task can be used for this purpose and these types of Tasks are often referred to as "request-response" Tasks for Choreography modelers.

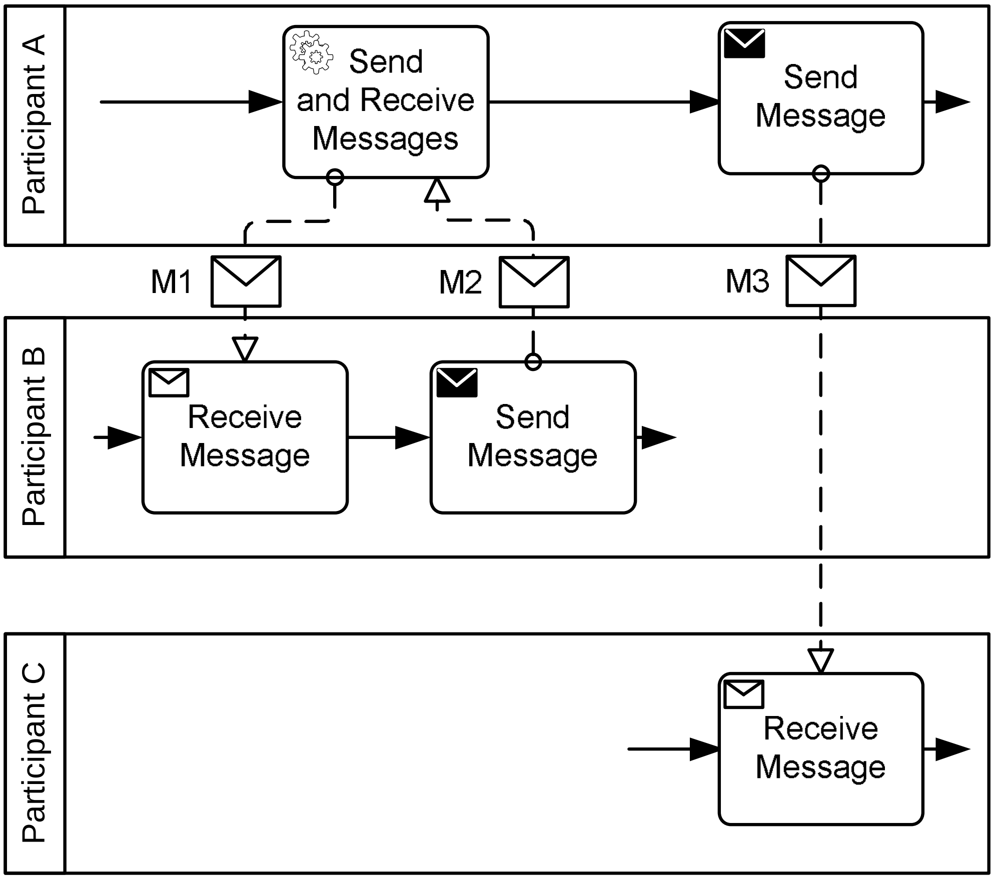


Figure 12-31 – The corresponding Collaboration for a valid Choreography sequence with a two-way Activity

Figure 12-32 shows how a sequence of Choreography Activities can be designed that would be invalid in the sense that an *Initiating Participant* would not know when the appropriate time would be to send a Message. In this example, "Participant A" is scheduled to send a Message to "Participant C" after "Participant B" sends a Message to "Participant C." However, "Participant A" will not know when the Message from "Participant B" has been sent. So, there is no way to enforce the sequence that is modeled in the Choreography.

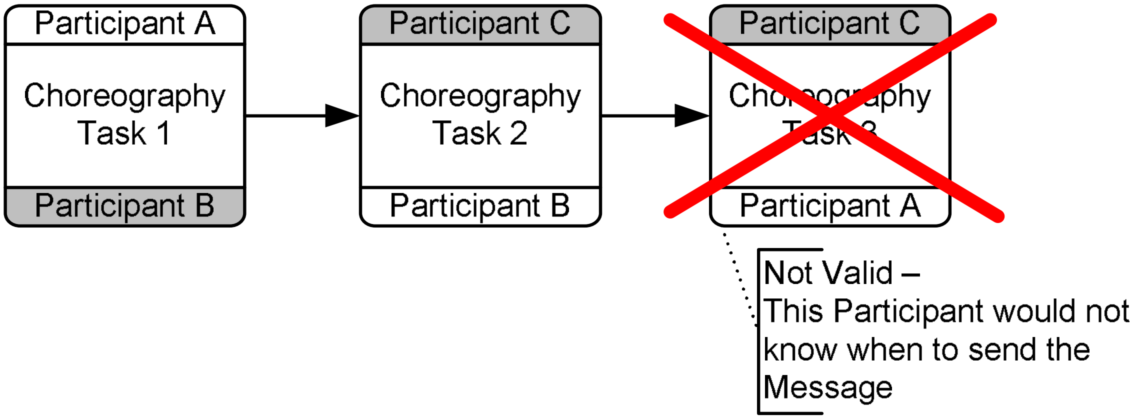


Figure 12-32 – An invalid sequence of Choreography Activities

Figure 12-33 shows the Collaboration view of the above Choreography diagram. It becomes clear that "Participant A" will not know the appropriate time to send Message "M3" to "Participant C." If the Message is sent too soon, then "Participant C" will not be prepared to receive it. Thus, as a Choreography, the model in Figure 12-32, above, cannot be enforced.

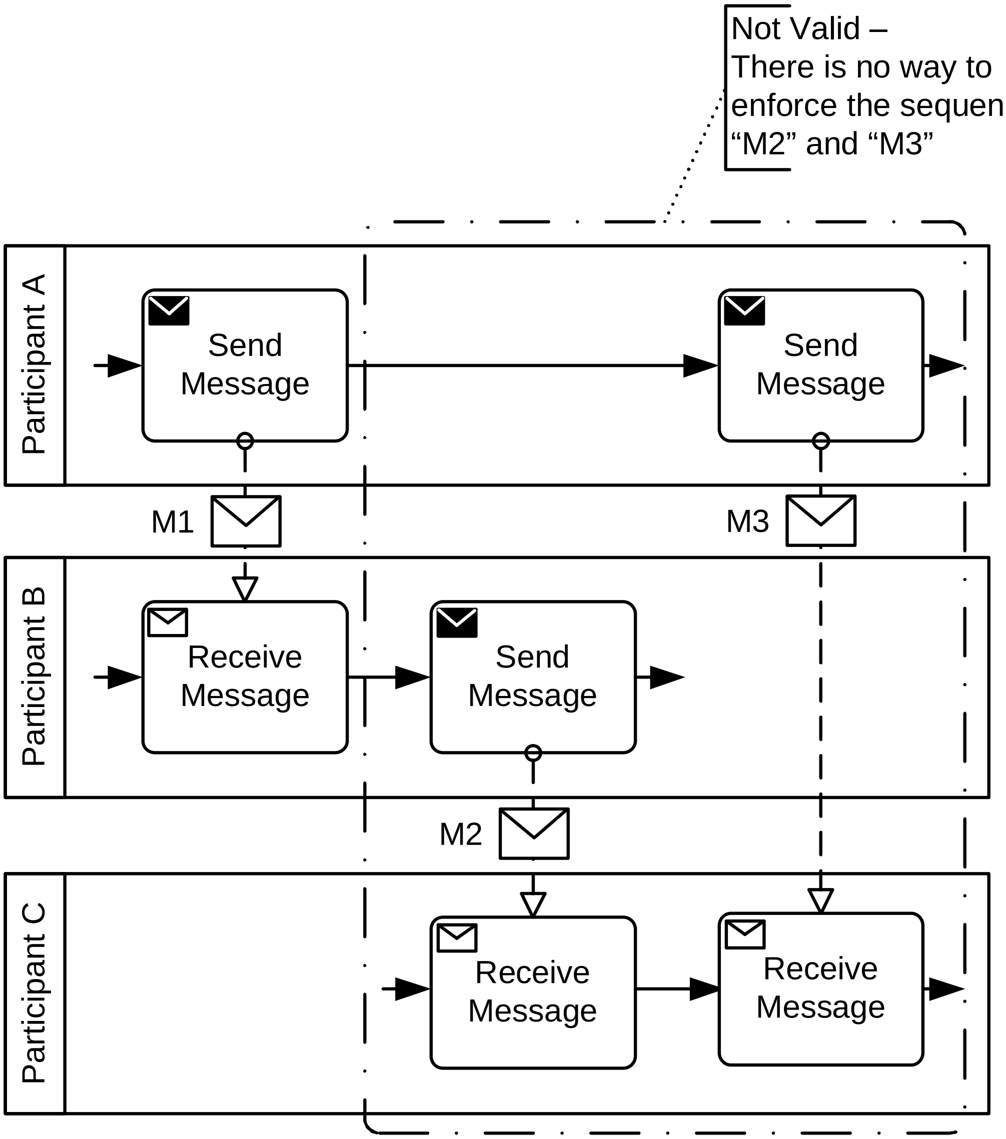


Figure 12-33 – The corresponding Collaboration for an invalid Choreography sequence

12.5. Events

12.5.1. Start Events

Start Events provide the graphical marker for the start of a Choreography. They are used much in the same way as they are used for a Process (see “Start Event” on page 244).

This table shows how the types of Start Events are applied to Choreography.

Table 12-6 – Use of Start Events in Choreography

Type of Event	Usage in Choreography?
None	**Yes**. This is really just a graphical marker since the arrival of the first Message in the Choreography is really the *Trigger* for the Choreography. Sub-Processes, however, we should look at. The *Parent* Process may be considered the Trigger. Not used in an Event Sub-Process.
Message	**No**. A Message Start Event, in a stand-alone Choreography, has no way to show who the senders or receivers of the Message should be. A Choreography Task should be used instead. Thus, a None Start Event should be used as a graphical marker for the "start" of the Choreography. Not used in an Event Sub-Process.
Timer	**Yes**. All *Participants* have to have an agreement to the approximate time.. Can be used in an Event Sub-Process.
Escalation	
Error	***No***. An Error is only visible to a single *Participant*. That *Participant* will have to send a Message to the other *Participants*.
Compensation	
Conditional	[Used only for Event Sub-Processes] **Yes**. This is actually determined internal to *Participant*, but then the other *Participants* know this has happened based the first interaction that follows.
Signal	**Yes**. The source of the *Signal* is not required (and may not even be a *Participant* in the Choreography). There are no specific recipients of a *Signal*. All *Participants* of the Choreography (to comply) must be able to see the *Signal*. Can be used in an Event Sub-Process.
Multiple	**Yes**. But they can only be Multiple *Signals or Timers*. As in *Orchestration*, this acts like an OR. Any one of the incoming *Signals* will Trigger the Choreography. Any *Signal* that follows would create a separate instance of the Choreography. Can be used in an Event Sub-Process.

12.5.2. Intermediate Events

Table 12-7 – Use of Intermediate Events in Choreography

Type of Event	Usage in Choreography?
None: in Normal Flow	**Yes**. However, this really doesn't have much meaning other than just documenting that a specific point has been reached in the Choreography. There would be no Message exchange or any delay in the Choreography.
None: Attached to Activity boundary	**No**. There would be no way for *Participants* to know when the Activity should be interrupted.
Message: in Normal Flow	**No**. A Message Intermediate Event, in a stand-alone Choreography, has no way to show who the senders or receivers of the Message should be. A Choreography Task should be used instead. Also, would the Event be a *Catch* or a *Throw*?
Message: Attached to Activity boundary	**Yes**. Only for Choreography Tasks. The Intermediate Event has to be attached to the Participant Band of the receiver of the Message (since it is a catch Event). The sender of the message has to be the other *Participant* of the Choreography Task.
Message: Use in Event Gateway	**No**. A Message Intermediate Event, in a stand-alone Choreography, has no way to show who the senders or receivers of the Message should be. A Choreography Task should be used instead.
Timer: in Normal Flow	**Yes**. Time is not precise in Choreography. It is established by the last visible Choreography Activity. The *Participants* involved in the Choreography Activity that immediately precedes will have a rough approximation of the time—there will be no exact synchronization. For relative timers: Only the *Participants* involved in the Choreography Activity that immediately precedes the Event would know the time. The sender of the Choreography Activity that immediately follows the timer must be involved in the Choreography Activity that immediately precedes the timer. For absolute timers (full time/date): All *Participants* would know the time. There does not have to be a relationship between the *Participants* of the Choreography Activities that are on either side the timer. The sender of the Choreography Activity that immediately follows the timer is the *Participant* that enforces the timer.

Timer: Attached to Activity boundary	**Yes**. Time is not exact in this case. It is established by the last visible Event. All *Participants* will have a rough approximation of the time—there will be no exact synchronization. This includes both interrupting and escalation Events. The *Participants* of the Choreography Activity that has the attached timer all enforce the timer. For relative timers: They all have to be involved in the Choreography Activity that immediately precedes the Activity with the attached timer. For absolute timers (full time/date): All *Participants* would know the time. They all have to be involved in the Choreography Activity that immediately precedes the Activity with the attached timer.
Timer: Used in Event Gateway	**Yes**. See Event-Based Gateway below.
Error: Attached to Activity boundary	***No***. An Error is only visible to a single *Participant*. That *Participant* will have to send a Message to the other *Participants*.
Escalation: Used in Normal Flow	
Escalation: Attached to Activity boundary	
Cancel: in Normal Flow	**No**. These are *Throw* Events. As with a Message Event, there would be no indicator as to who is the source of the *Cancel*.
Cancel: Attached to Activity boundary	**Yes**. These are *Catch* Events. As with a Message Event, they would be attached to the Choreography Activity on the Participant Band that is receiving the Cancel. These would only be interrupting Events.
Compensation: in Normal Flow	**No**. These are *Throw* Events. As with a Message, there would be no indicator as to who is the source of the *Cancel*.
Compensation: Attached to Activity boundary	**Yes**. These are *Catch* Events. As with a Message Event, they would be attached to the Choreography Activity on the Participant Band that is receiving the *Cancel*.
Conditional: in Normal Flow	**Yes**. This is a delay that waits for a change in data to trigger the Event. The data must be visible to the *Participants* as it was data of a previously sent Message.
Conditional: Attached to Activity boundary	**Yes**. This is an interruption that waits for a change in data to trigger the Event. The data must be visible to the *Participants* as it was data of a previously sent Message.
Conditional: Used in Event Gateway	**Yes**. This is a delay that waits for a change in data to trigger the Event. The data must be visible to the *Participants* as it was data of a previously sent Message.

Link: in Normal Flow	**Yes**. These types of Events merely create a virtual Sequence Flow. Thus, as long as a Sequence Flow between two elements is valid (and within a Choreography Process level), then a pair of Link Events can interrupt that Sequence Flow.
Signal: in Normal Flow	**Yes**. Only *Catch* Events can be used. For *Throw* Signal Events, there would be no indicator of who is the source *Participant*. This would be a delay in the Choreography that waits for the *Signal*. The source of the *Signal* is not required (and may not even be a *Participant* in the Choreography). There are no specific recipients of a *Signal*. All *Participants* of the Choreography (to comply) must be able to see the *Signal*.
Signal: Attached to Activity boundary	**Yes**. These are *Catch* Events. This would be an interruption in the Choreography that waits for the *Signal*. The source of the *Signal* is not required (and may not even be a *Participant* in the Choreography). There are no specific recipients of a *Signal*. All *Participants* of the Choreography (to comply) must be able to see the *Signal*. This Event should not (must not?) be attached to a Participant Band or this would suggest that that *Participant* is a specific recipient of the *Signal*.
Signal: Used in Event Gateway	**Yes**. These are *Catch* Events. This would be a delay in the Choreography that waits for the *Signal*. The source of the *Signal* is not required (and may not even be a *Participant* in the Choreography). There are no specific recipients of a *Signal*. All *Participants* of the Choreography (to comply) must be able to see the *Signal*.
Multiple: in Normal Flow	**Yes**. But they can only be a collection of valid *Catch* Events. As in *Orchestration*, this acts like an OR. Any one of the incoming triggers will continue the Choreography.
Multiple: Attached to Activity Boundary	**Yes**. But they can only be a collection of valid *Catch* Events. As in *Orchestration*, this acts like an OR. Any one of the incoming triggers will interrupt the Choreography Activity.

12.5.3. End Events

End Events provide a graphical marker for the end of a path within the Choreography.

Table 12-8 – Use of End Events in Choreography

Type of Event	Usage in Choreography?
None	**Yes**. This is really just a graphical marker since the sending of the previous Message in the Choreography is really the end of the Choreography. The *Participants* of the Choreography would understand that they would not expect any further Message at that point.
Message	**No**. A Message End Event, in a stand-alone Choreography, has no way to show who the senders or receivers of the Message should be. A Choreography Task should be used instead. Thus, a None End Event should be used as a graphical marker for the "end" of the Choreography
Error	**No**. These are *Throw* Events and there would be no way to indicate the *Participant* that is the source of the Error.
Escalation	**No**. These are *Throw* Events and there would be no way to indicate the *Participant* that is the source of the Escalation..
Cancel	**No**. These are *Throw* Events. As with a Message Event, there would be no indicator as to who is the source of the *Cancel*.
Compensation	**No**. These are *Throw* Events. As with a Message Event, there would be no indicator as to who is the source of the *compensation*.
Signal	**No**. These are *Throw* Events. As with a Message Event, there would be no indicator as to who is the source of the *Signal*.
Multiple	**No**. Since there are no valid End Event Results (Terminate doesn't count) in Choreography, there cannot be multiple of them.
Terminate	**Yes**. However, there would be no specific ability to terminate the Choreography, since there is no controlling system. In this case, all *Participants* in the Choreography would understand that when the Terminate End Event is reached (actually when the Message that precedes it occurs), then no further messages will be expected in the Choreography, even if there were parallel paths. The use of the Terminate End Event really only works when there are only two (2) *Participants*. If there are more than two (2) *Participants*, then any *Participant* that was not involved in the last Choreography Task would not necessarily know that the Terminate End Event had been reached.

12.6. Gateways

In an *Orchestration* Process, Gateways are used to create alternative and/or parallel paths for that Process. Choreography has the same requirement of alternative and parallel paths. That is, interactions between *Participants* may happen in sequence, in parallel, or through exclusive selection. While the paths of Choreography follow the same basic patterns as that of an *Orchestration* Process, the lack of a central mechanism to maintain data visibility, and that there is no central evaluation, there are constraints as to how the Gateways are used in conjunction with the Choreography Activities that precede and follow the Gateways. These constraints are an extension of the basic sequencing constraints that was defined on page 364. The six (6) sections that follow will define how the types of Gateways are used in Choreography.

12.6.1. Exclusive Gateway

Exclusive Gateways (Decisions) are used to create alternative paths within a Process or a Choreography. For details of how Exclusive Gateways are used within an *Orchestration* Process see page 298.

Exclusive Gateways are used in Choreography, but they are constrained by the lack of a central mechanism to store the data that will be used in the *Condition* expressions of the Gateway's *outgoing* Sequence Flow. Choreographies may contain natural language descriptions of the Gateway's *Conditions* to document the alternative paths of the Choreography (e.g., "large orders" will go down one path while "small orders" will go down another path), but such Choreographies would be underspecified and would not be *enforceable*. To create an *enforceable* Choreography, the Gateway *Conditions* must be formal *Condition Expressions*; however:

- The data used for Gateway *Conditions* MUST have been in a Message that was sent prior to (upstream from) the Gateway.
    - More specifically, all *Participants* that are directly affected by the Gateway MUST have either sent or received the Message(s) that contained the data used in the *Conditions*.
        - Furthermore, all these *Participants* MUST have the same understanding of the data. That is, the actual values of the data cannot selectively change after a *Participant* has seen a Message. Changes to data during the course of the Choreography MUST be visible to all the *Participants* affected by the Gateway.

These constraints ensure that the *Participants* in the Choreography understand the basis (the actual value of the data) for the decision behind the Gateway.

One (1) or more *Participants* will actually "control" the Gateway decision; that is, these *Participants* make the decision through the internal *Orchestration* Processes. The decision is manifested by the particular Message that occurs in the Choreography (after the Gateway). This Message is the *initiating* Message of a Choreography Activity that follows the Gateway. Thus, only the *Participants* that are the *initiators* of the Messages that follow the Gateway are the ones that control the decision. This means that:

- The *initiating Participants* of the Choreography Activities that follow the Gateway MUST have sent or received the Message that provided the data upon which the decision is made.
    - The Message that provides the data for the Gateway MAY be in any Choreography Activity prior to the Gateway (i.e., it does not have to immediately precede Gateway).

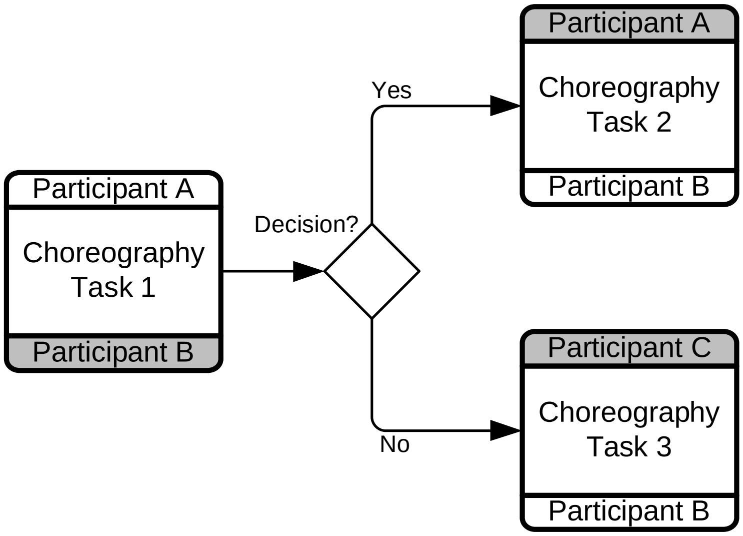


Figure 12-34 – An example of the Exclusive Gateway

Figure 12-35 shows the Collaboration that demonstrates how the above Choreography that includes an Exclusive Gateway can be enforced.

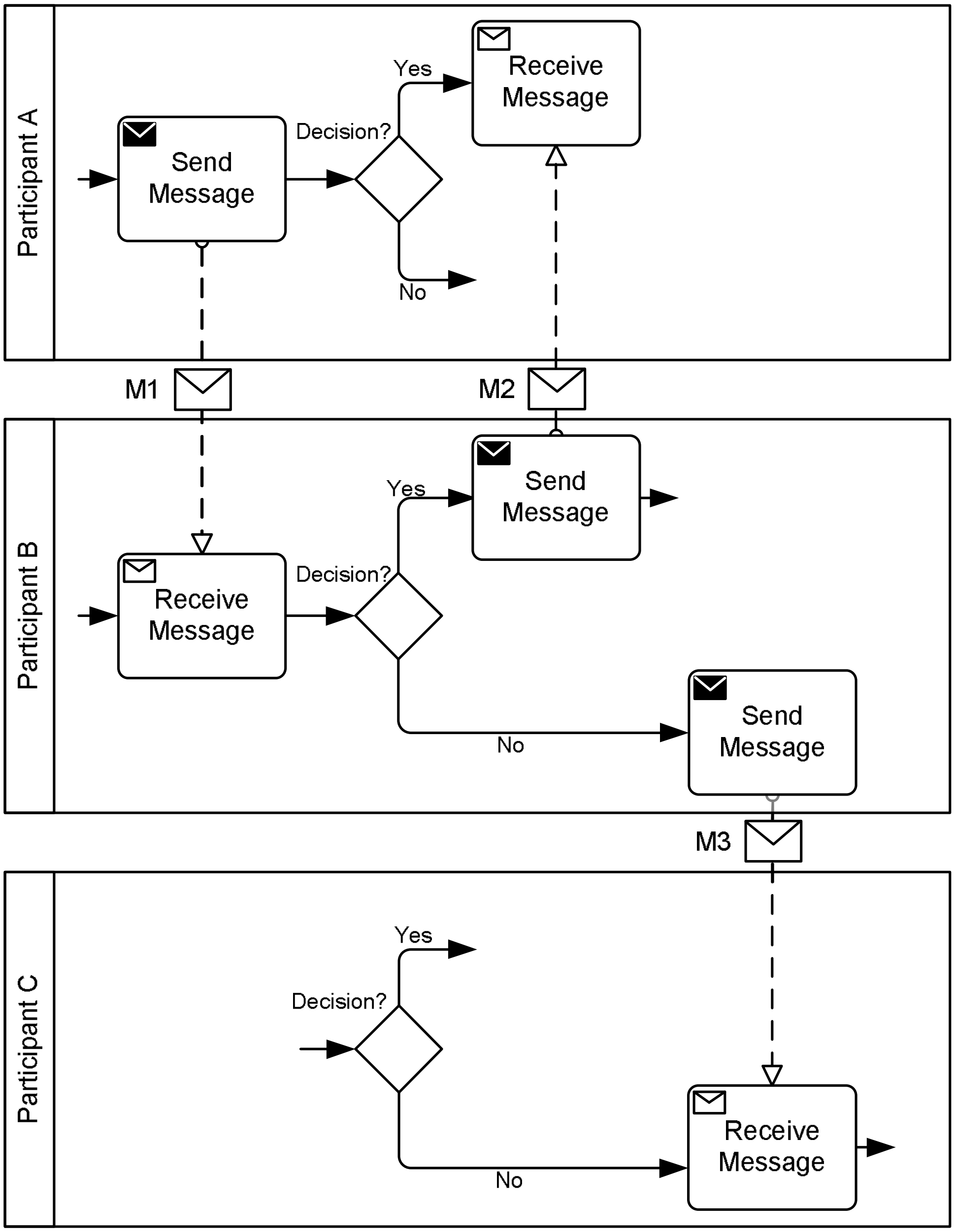


Figure 12-35 – The relationship of Choreography Activity *Participants* across the sides of the Exclusive Gateway shown through a Collaboration

Usually, the *initiators* for the Choreography Activities that follow the Gateway will be the same *Participant*. That is, there is only one (1) *Participant* controlling the decision. Often, the receivers of the *initiating* Message for those Choreography Activities will be the same *Participant*. However, it is possible that there could be different *Participants* receiving the *initiating* Message for each Choreography Activity (see Figure 12-36).

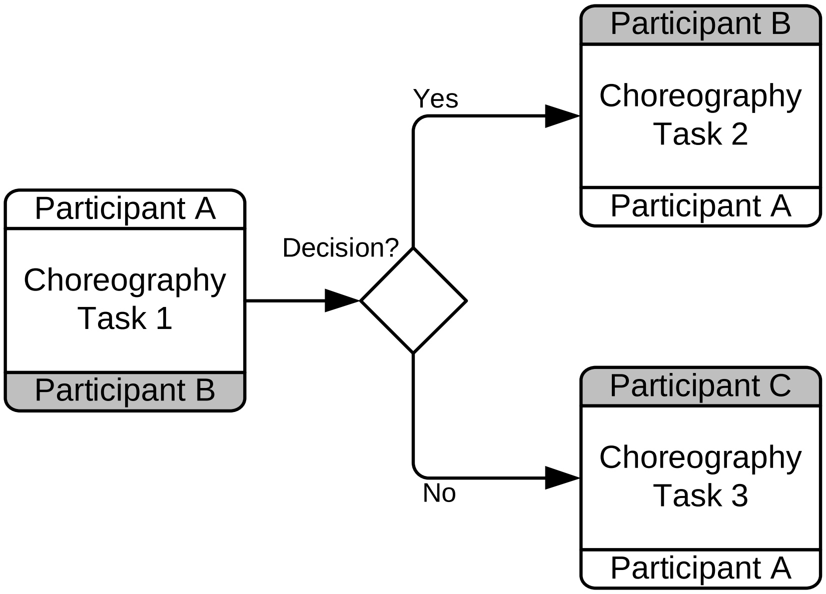


Figure 12-36 – Different *Receiving* Choreography Activity *Participants* on the output sides of the Exclusive Gateway

This configuration can only be valid if all the *Participants* in the Choreography Activities that follow the Gateway have seen the data upon which the decision is made. If either "Participant B" or "Participant C" had not sent or receive a Message with the appropriate data, then that *Participant* would not be able to know if they are suppose to receive a Message at that point in the Choreography. There is also the assumption that the value of the data is consistent from the point of view of all *Participants*.

Figure 12-37 displays the corresponding Collaboration view of the above Choreography Exclusive Gateway configuration.

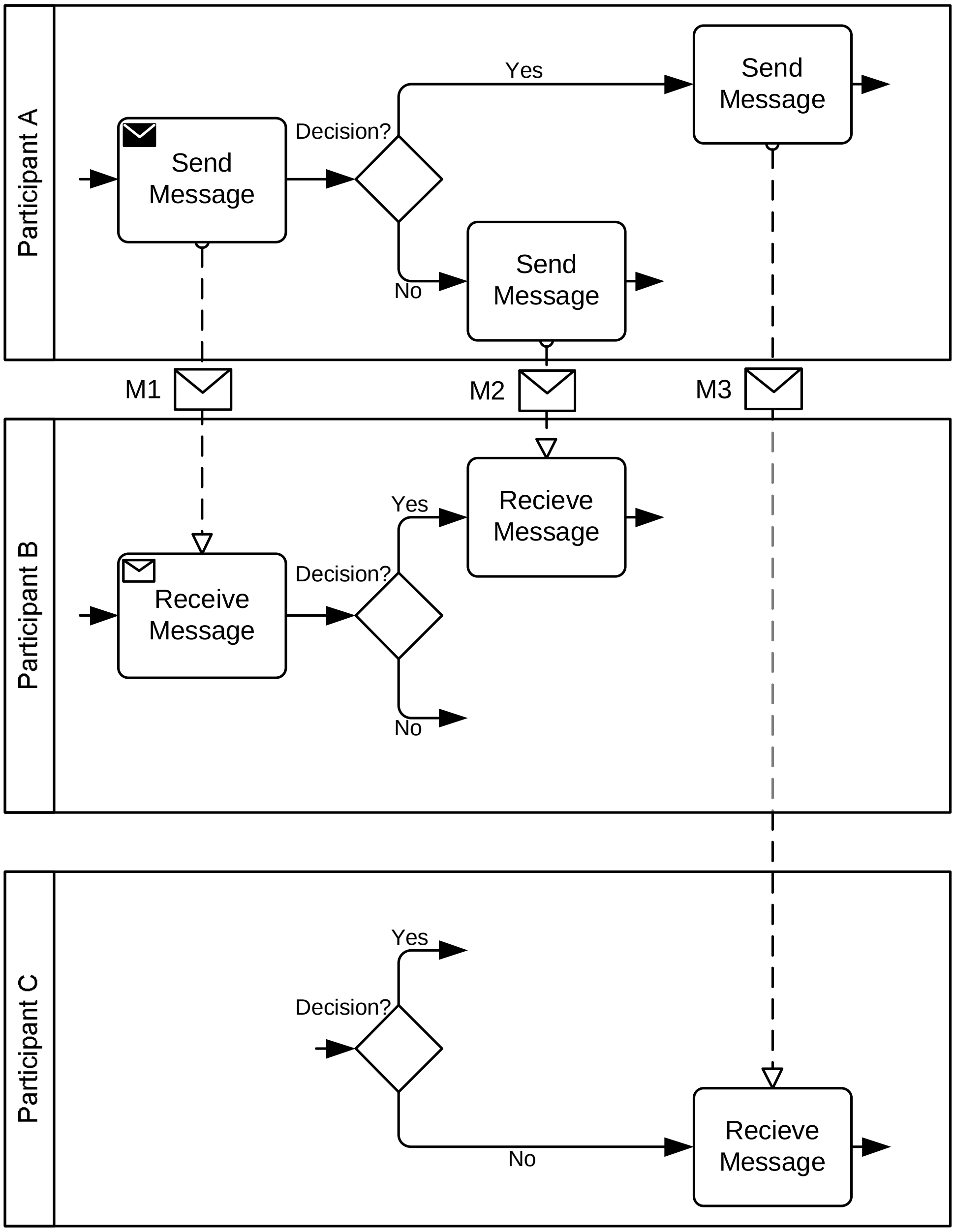


Figure 12-37 – The corresponding Collaboration view of the above Choreography Exclusive Gateway configuration

The required execution behavior of the Gateway and associated Choreography Activities are enforced through the Business Processes of the *Participants* as follows:

- Each Choreography Activity and the Sequence Flow connections are reflected in each *Participant* Process.
- The Gateway is reflected in the Process of each *Participant* Process that is an initiator of Choreography Activities that follow the Gateway
- For the receivers in Choreography Activities that follow the Gateway, an Event-Based Gateway is used to consume the associated Message (sent as an outcome of the Gateway). When

a *Participant* is the receiver of more than one of the alternative Messages, the corresponding receives follow the Event-Based Gateway. If the *Participant* is the receiver of only one such Message, that is also consumed through a receive following the Event-Based Gateway. This is because the *Participant* Process does not know whether it will receive a Message (since the Gateway entails a choice of outcomes).

12.6.2. Event-Based Gateway

As described above, the Event-Based Gateway represents a branching point in the Process where the alternatives are based on Events that occur at that point in the Process, rather than the evaluation of expressions using Process data. For details of how Event-Based Gateways are used within an *Orchestration* Process see Section "Event-Based Gateway" on page 307.

These Gateways are used in Choreography when the data used to make the decision is only visible to the internal Processes of one *Participant*. That is, there has been no Message sent within the Choreography that would expose the data used to make the *decision*. Thus, the only way that the other *Participants* can be aware of the results of the decision is by the particular Message that arrives next.

- On the right side of the Gateway: either
  - The senders MUST to be the same; or
  - The receivers MUST to be the same
    - After the first Choreography Activity occurs, the other Choreography Activities for the Gateway MUST NOT occur.
- Message Intermediate Events MUST NOT be used in the Event-Based Gateway.
- Timer Intermediate Events MAY be used, but they restrict the participation in the Gateway.
  - For relative timers: All *Participants* on the right side of the Gateway MUST be involved in the Choreography Activity that immediately precedes the Gateway.
  - For absolute timers (full time/date): All *Participants* on the right side of the Gateway MUST be involved in the Choreography Activity that immediately precedes the Gateway.
- Signal Intermediate Events MAY be used (they are visible to all *Participants*)
- No other types of Intermediate Events are allowed.

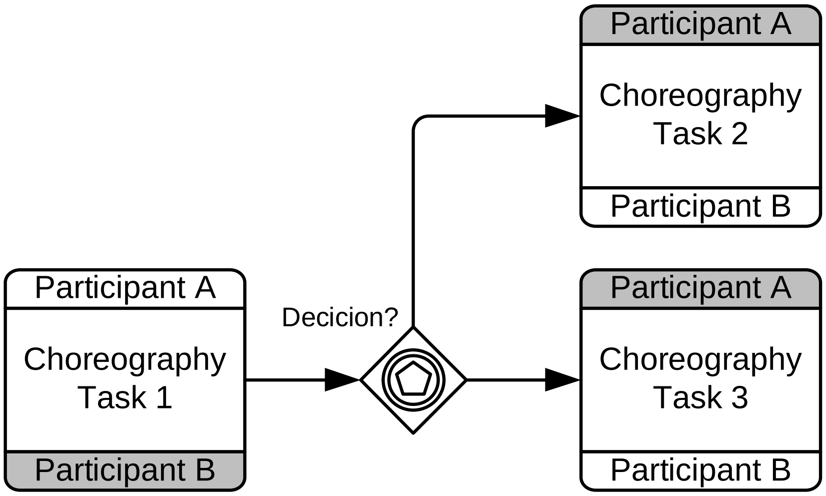


Figure 12-38 – An example of an Event Gateway

Figure 12-39 displays the corresponding Collaboration view of the above Choreography Event Gateway configuration.

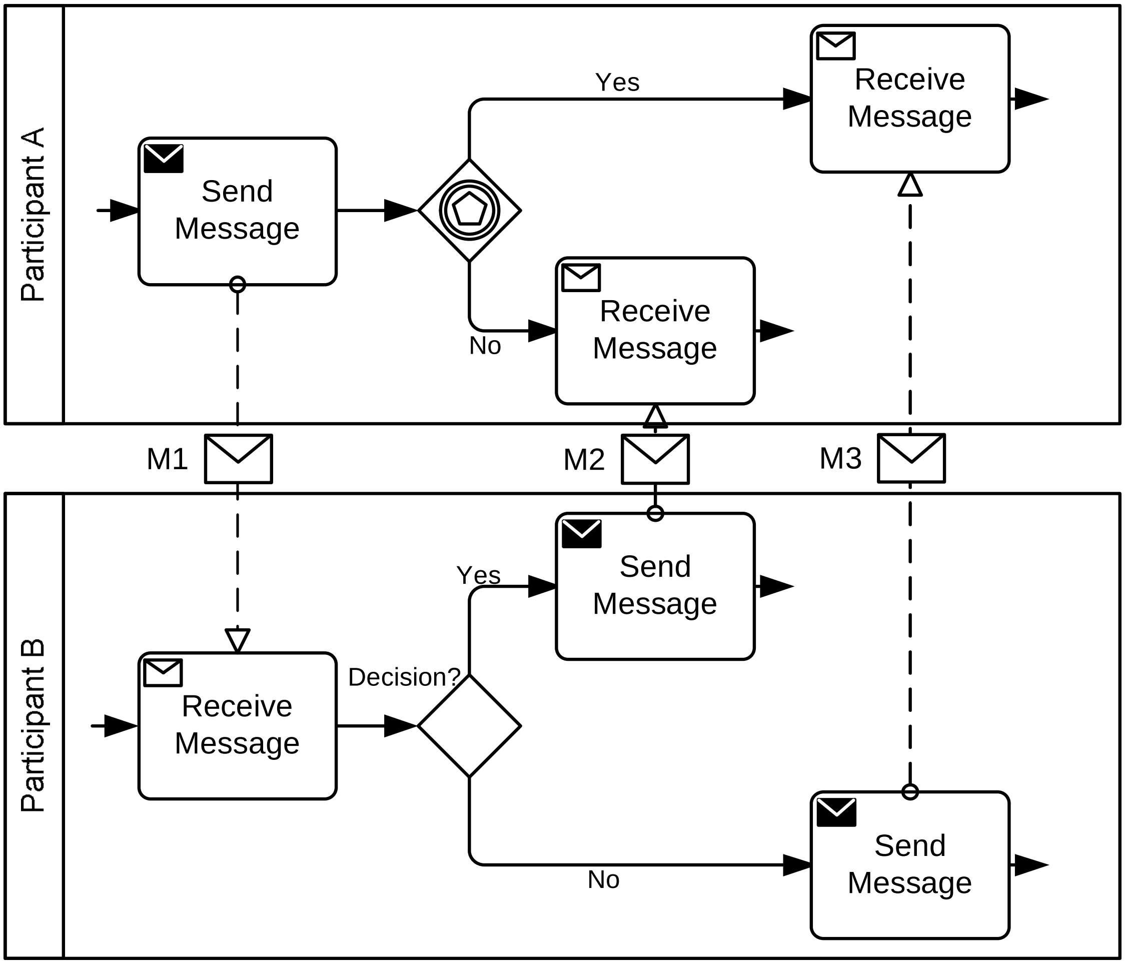


Figure 12-39 – The corresponding Collaboration view of the above Choreography Event Gateway configuration

The required execution behavior of the Event-Based Gateway and associated Choreography Activities are enforced through the Business Processes of the *Participants* as follows:

- Each Choreography Activity and the Sequence Flow connections is reflected in each *Participant* Process.
- If the senders following the Gateway are the same, the Event-Based Gateway is reflected as an Exclusive Gateway in that *Participant's* Process. This is because the choice of which Message to send is determined by the same Participant. If the senders are different, sending occurs through different Processes.
- If the receivers are the same, the senders can be the same or different. In this case, the Event-Based Gateway is reflected in the receiver's Process, with the different Message receives following the Gateway.
- If the receivers are different, the senders need to be the same. The Event-Based Gateway is reflected for different receiver Processes such that the respective receive follows the Gateway. A time-out may be used to ensure that the Gateway does not wait indefinitely.

12.6.3. Inclusive Gateway

Inclusive Gateways are used for modeling points of synchronization of a number of branches, not all of which are active, after which one or more alternative branches are chosen within a Choreography flow. For example, one of more branches may be activated upstream, in parallel, depending on the nature of goods in an order (e.g. large orders, fragile goods orders, orders belonging to pre-existing shipment contracts), and these are subsequently merged. The point of merge results in one or more risk mitigating outcomes (e.g. special insurance protection needed, special packaging needed, and different container categories needed). Inclusive Gateways are also used within an *Orchestration* Process see page 300.

Like Exclusive Gateways, Inclusive Gateways are used in a Choreography, but they are constrained by the lack of a central mechanism to store the data that will be used in the *Condition* expressions of the Gateway's *outgoing* Sequence Flows. Choreographies may contain natural language descriptions of the Gateway's *Conditions* to document the one more alternative paths of the Choreography (e.g., "special insurance protection needed", "special packaging needed", and different "container category needed"), but such Choreographies would be underspecified and would not be *enforceable*. To create an *enforceable* Choreography, the Gateway *Conditions* must be formal *Condition Expressions*. In general the following rules apply for the *Expressions:*

Like the enforceability of the Exclusive Gateway, the Inclusive Gateway in a Choreography requires that the data in the *Expressions* of the outgoing Sequence Flow of the Gateway be available to the initiators of the Choreography Activities of *outgoing* Sequence Flow. This means that the initiators of these Choreography Activities should also be senders or receivers of Messages in Choreography Activities immediately preceding the Gateway. The major difference, however, is that the synchronizing behavior of the Inclusive Gateway can only be enforced through one participant. Hence, the rules for enforceability are as follows:

- ◆ The data used for Gateway *Conditions* MUST have been in a Message that was sent prior to (upstream from) the Gateway.
  - ◆ More specifically, all *Participants* that are directly affected by the Gateway MUST have either sent or received the Message(s) that contained the data used in the *Conditions*.
    - ◆ Furthermore, all these *Participants* MUST have the same understanding of the data. That is, the actual values of the data cannot selectively change after a *Participant* has seen a Message. Changes to data during the course of the Choreography MUST be visible to all the *Participants* affected by the Gateway.
  - ◆ Merge: In order to enforce the synchronizing merge of the Gateway, the senders or receivers Choreography Activities preceding the Gateway need to be the same *Participant*. This ensures that the merge can be enforced. (This relies on the assumption of logical atomicity of a Choreography Activity, otherwise the rule would require that all receivers are the same so that the Gateway is enforced in the receiver's Process only).
- ◆ Split: In order to enforce the split side of the Gateway, the initiators of all Choreography Activities immediately following the Gateway must be the same as the common sender or receiver of Choreography Activities preceding the Gateway. The sender(s) of all the Choreography Activities after the Gateway must be involved in all the Choreography Activities that immediately precede the Gateway.

Figure 12-40 shows an example of a Choreography with an Inclusive Gateway. The Gateway is enforced in the corresponding Business Processes of the *Participants* involved. For the merge behavior to be enforced, the receivers of Choreography Activities preceding the Gateway and the initiator of Choreography Activities immediately following the Gateway are the same Participant (i.e. B).

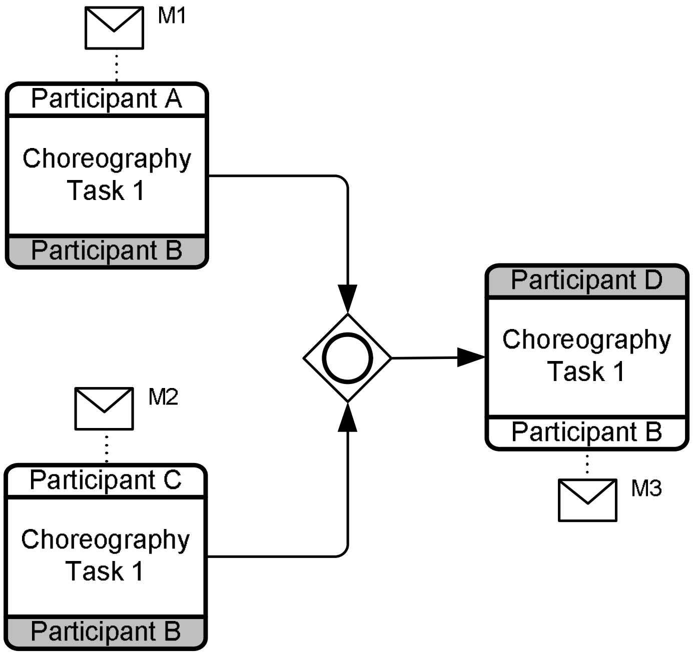


Figure 12-40 – An example of a Choreography Inclusive Gateway configuration

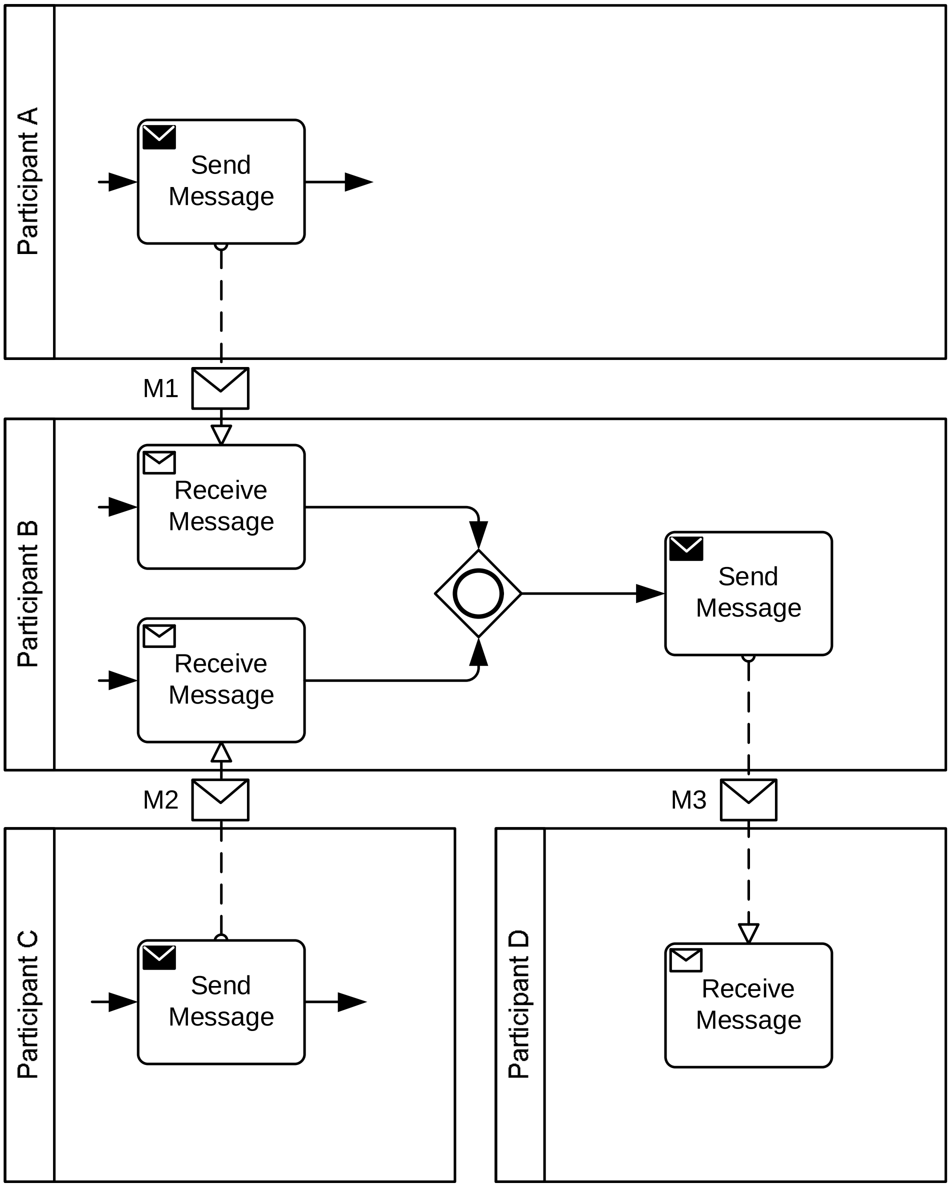


Figure 12-41 – The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration

Figure 12-42, a variation of Figure 12-40 above. shows an example of a Choreography illustrating the enforcement of the split behavior of the Inclusive Gateway. For the split behavior to be enforced, the initiators of Choreography Activities immediately following the Gateway and the receiver of Choreography Activities immediately preceding the Gateway are the same Participant (i.e. A).

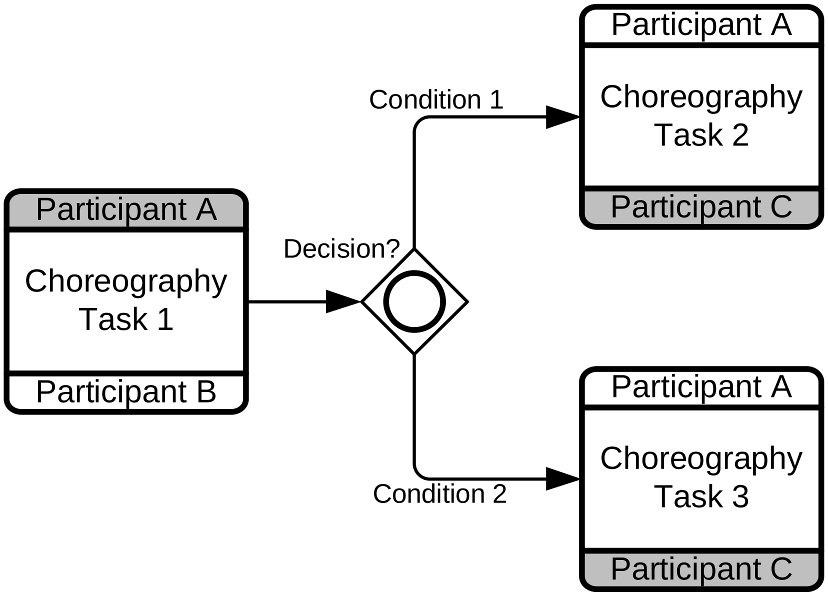


Figure 12-42 – An example of a Choreography Inclusive Gateway configuration

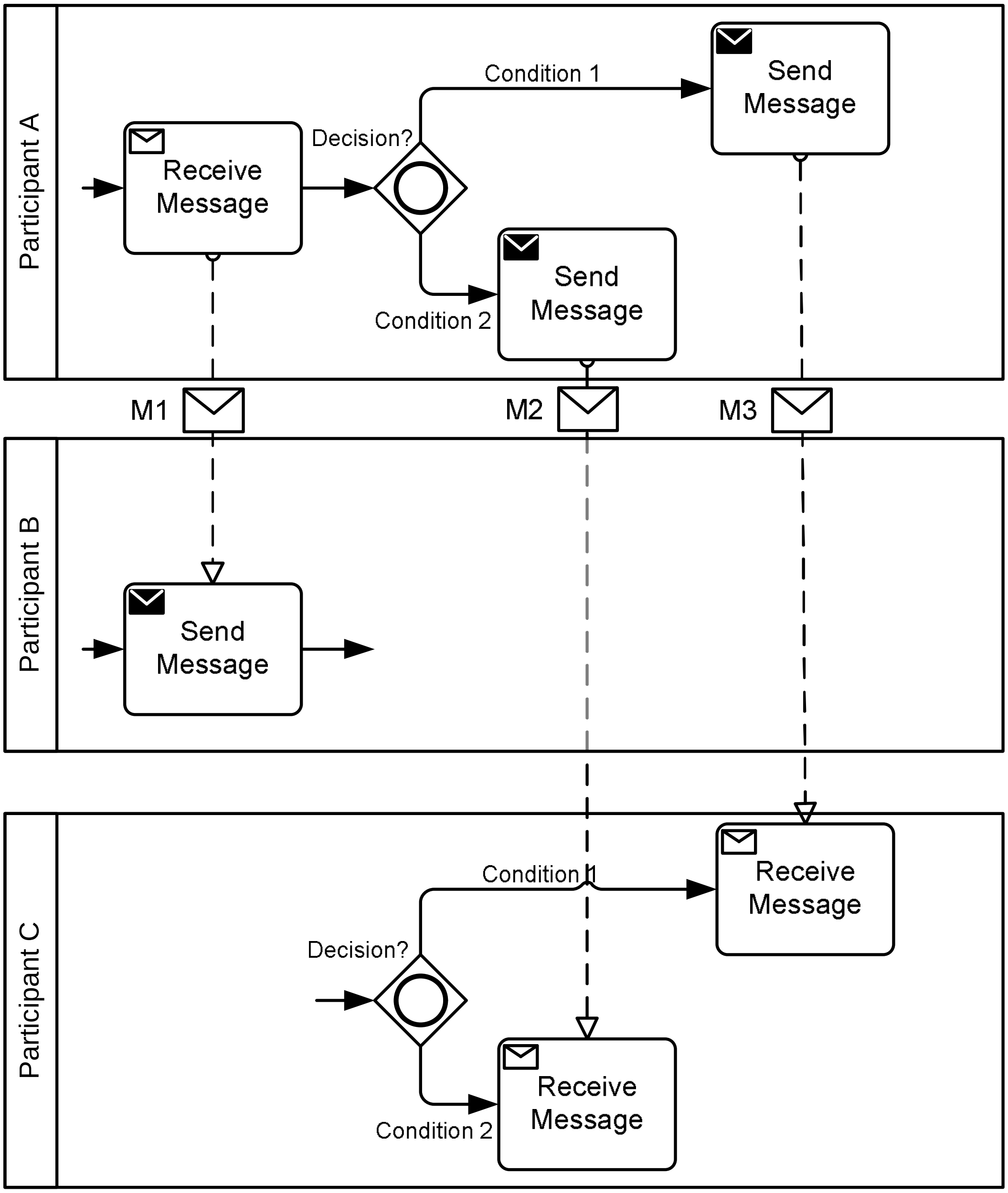


Figure 12-43 – The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration

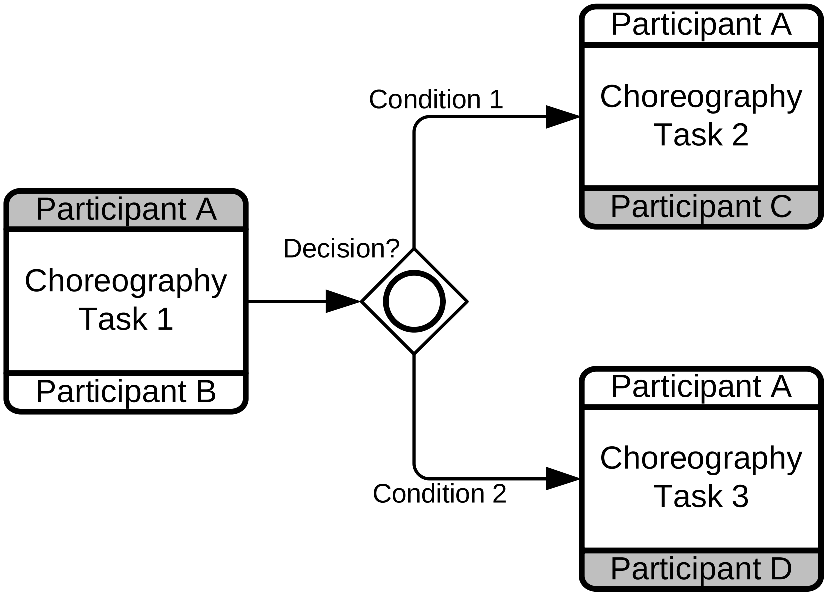


Figure 12-44 – Another example of a Choreography Inclusive Gateway configuration

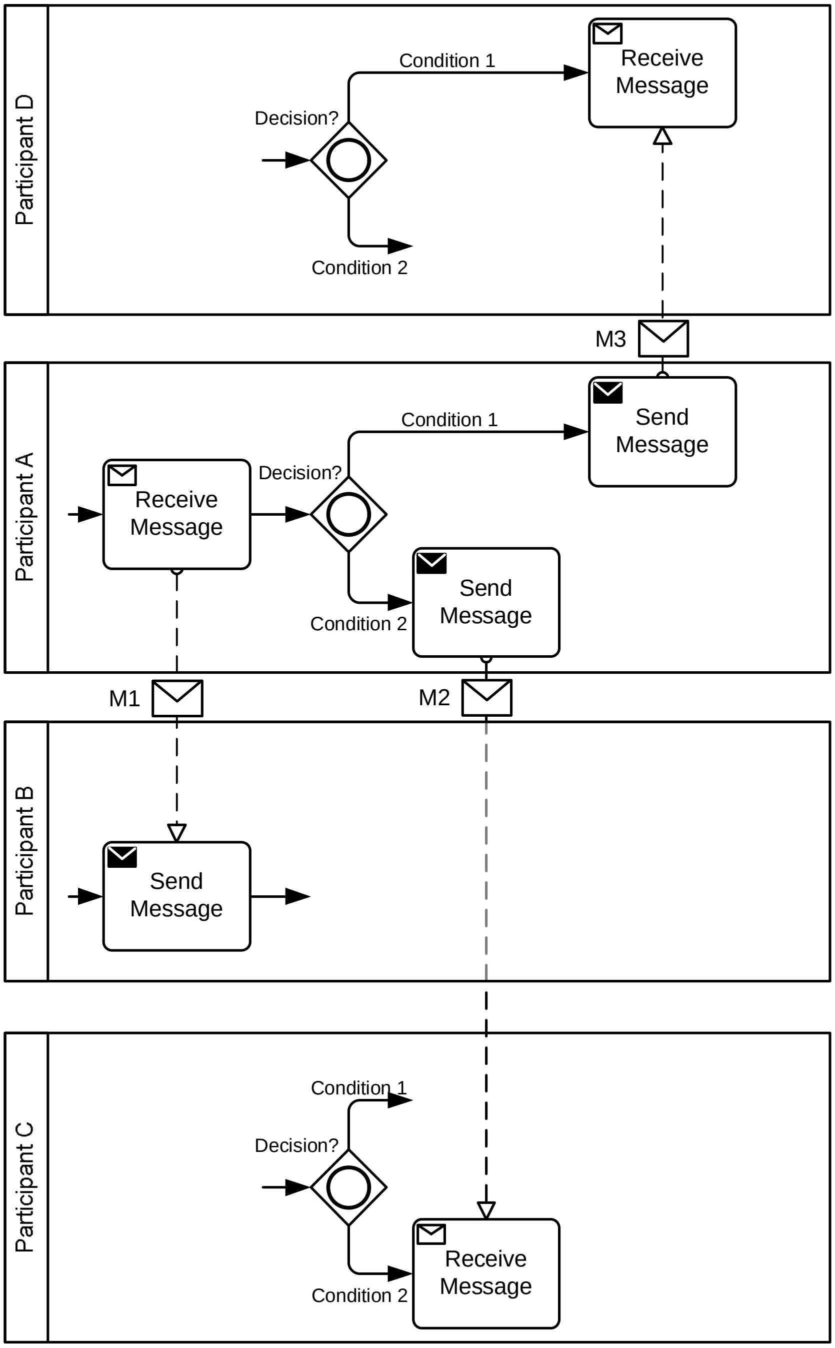


Figure 12-45 – The corresponding Collaboration view of the above Choreography Inclusive Gateway configuration

12.6.4. Parallel Gateway

Parallel Gateways are used to create paths and are performed at the same time, within a Choreography flow. For details of how Parallel Gateways are used within an *Orchestration* Process see page 302.

Since there is no conditionality for these Gateways, they are available as-is in Choreography. They create parallel paths of the Choreography that all *Participants* are aware of.

- The sender(s) of all the Activities after the Gateway must be involved in all the Activities that immediately precede the Gateway.
  - If there is a chain of Gateways with no Choreography Activities in between, the Choreography Activity that precedes the chain satisfies the above constraint.

Figure 12-46 shows the relationship of Choreography Activity *Participants* across the sides of the Parallel Gateway.

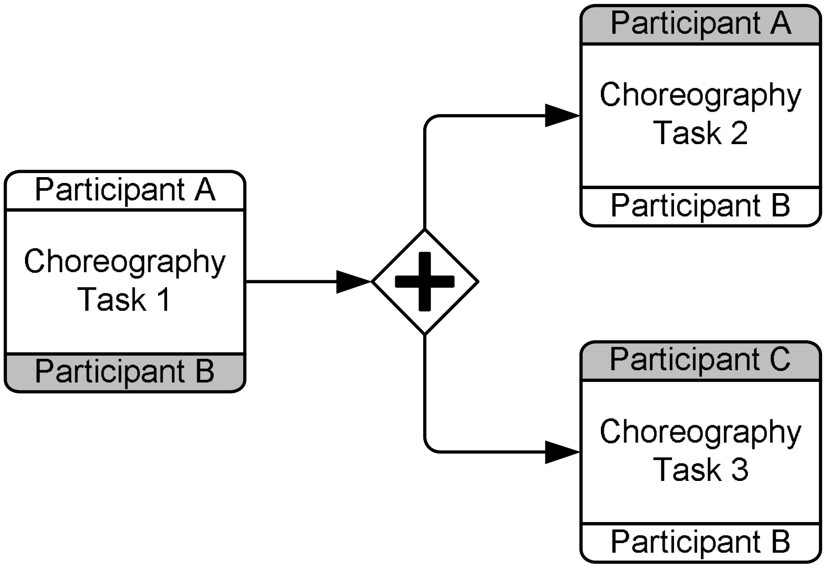


Figure 12-46 – The relationship of Choreography Activity *Participants* across the sides of the Parallel Gateway

Figure 12-47 shows the corresponding Collaboration view of the above Choreography Parallel Gateway configuration.

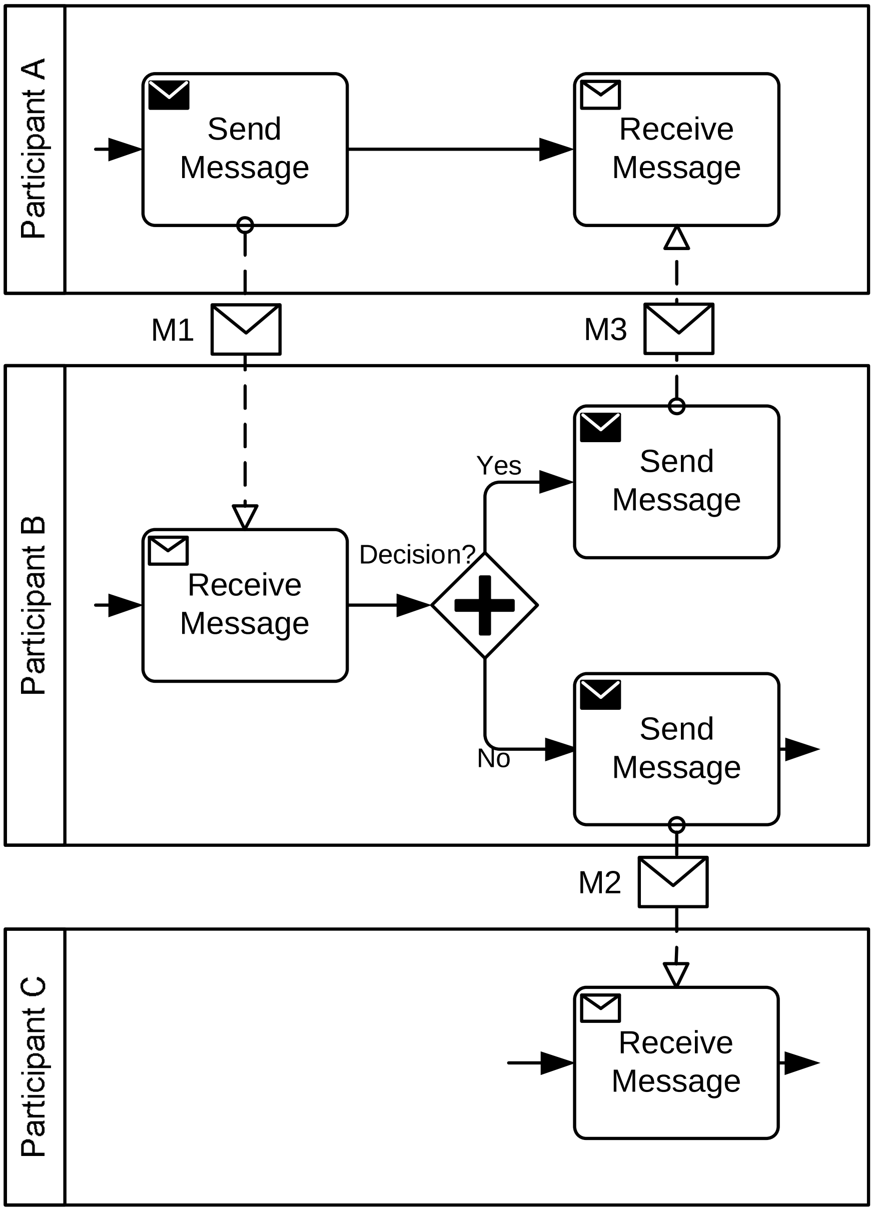


Figure 12-47 – The corresponding Collaboration view of the above Choreography Parallel Gateway configuration

The required execution behavior of the Parallel Gateway and associated Choreography Activities are enforced through the Business Processes of the *Participants* as follows:

- Each Choreography Activity and the Sequence Flow connections is reflected in each *Participant* Process.
- If the senders following the Parallel Gateway are the same, a Parallel Gateway is reflected in the sender's Process followed by Message sending actions to the corresponding receivers
- If the senders are different, the Parallel Gateway is manifested by Sequence Flows followed by the sending action in each Process.

12.6.5. Complex Gateway

Complex Gateways can model partial merges in Business Processes where when some but not all of a set of preceding branches complete, the Gateway fires. This can be considered the discriminator/n-of-m join pattern[2] and is not supported through the inclusive OR merge since it is not concerned with sets of branches, but rather branches which have *tokens*. Applied in Choreographies, Complex Gateways can model tendering and information canvassing use cases where requests are sent to participants who respond at different times.

Consider an e-tender which sends a request for quote to service providers (e.g. warehouse storage) in a marketplace. The e-tender Process sends out each request and anticipates a response through two Choreography Activities with a sequential flow between these. The request-response branches merge at a Complex Gateway to model the requirement that when 60% responses have arrived, an assessment of the tender can proceed. The assessment occurs after the Complex Gateway. If the assessment reports that the reserve amount indicated by the customer cannot be met, a new iteration of the tender is made. All up a maximum of 3 tenders is run. A key issue is to ensure that the responses should not be mixed across tender iterations.

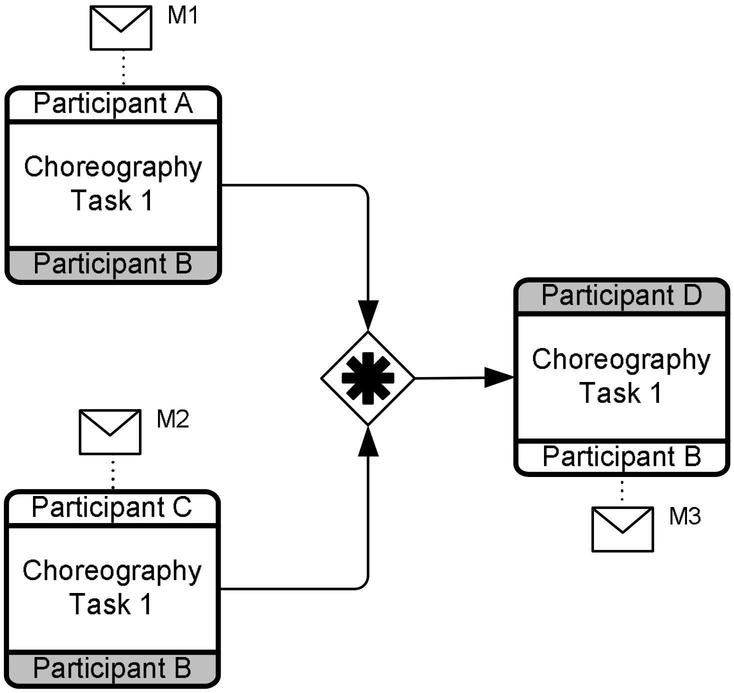


Figure 12-48 – An example of a Choreography Complex Gateway configuration

[2] http://www.workflowpatterns.com/patterns/control/advanced_branching/wcp9.php

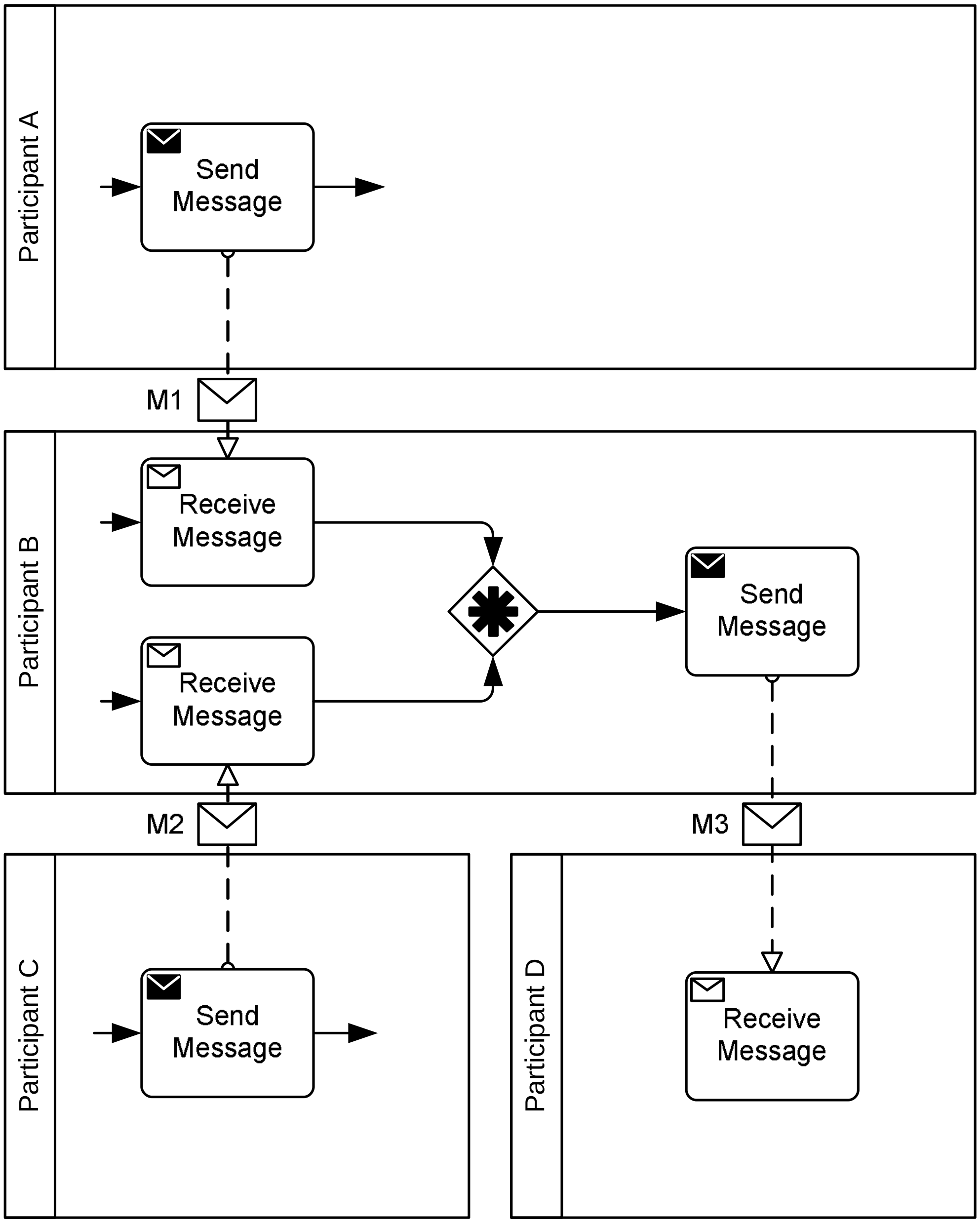


Figure 12-49 – The corresponding Collaboration view of the above Choreography Complex Gateway configuration

12.6.6. Chaining Gateways

It is possible to chain Gateways. This means that a modeler can sequence two (2) or more Gateways without any intervening Choreography Activities, however the constraints on what participants can appear before and after the chain must be observed.

12.7. Choreography within Collaboration

Participants

Participants are used in both Collaborations and Choreographies.

Swimlanes

Swimlanes, both Pools and Lanes, are not used in Choreographies. Pools are used exclusively in Collaborations (see page 146). *Participants*, which can be associated to Pools, however, are used in the *Participant Bands* of Choreography Tasks (see page 350) and Choreography Sub-Processes (see page 356). Pools can be used with Choreography diagrams when in the context of a Collaboration diagram (see page 394).

Lanes are not used in Choreography diagrams since Lanes are sub-partitions of a Pool and Choreographies are placed in between the Pools (if used in a Collaboration).

Figure 12-50 shows an example of a Choreography Process combined with Black Box Pools.

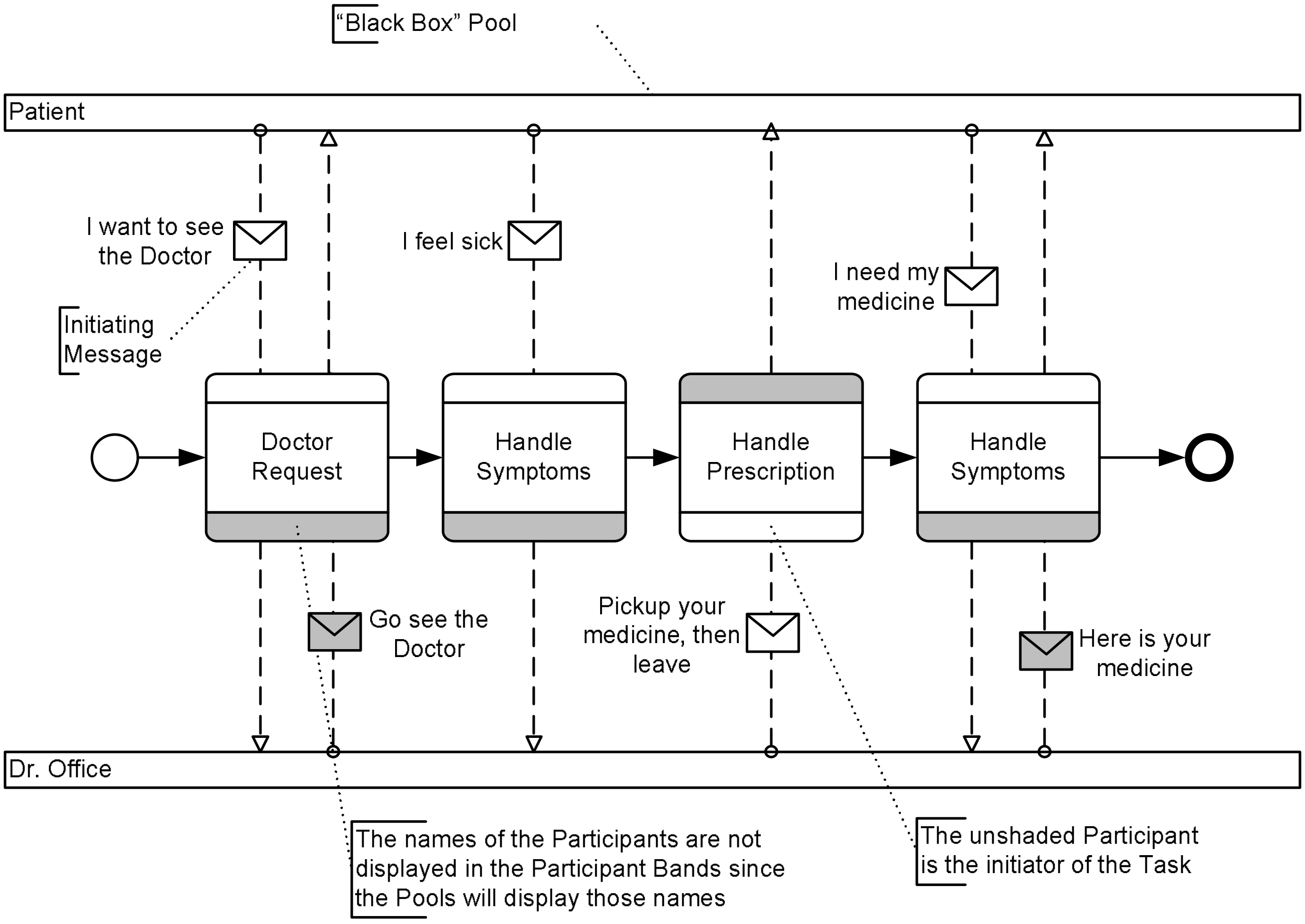


Figure 12-50 – An example of a Choreography Process combined with Black Box Pools

Figure 12-51 shows an example of a Choreography Process combined with Pools that contain Processes.

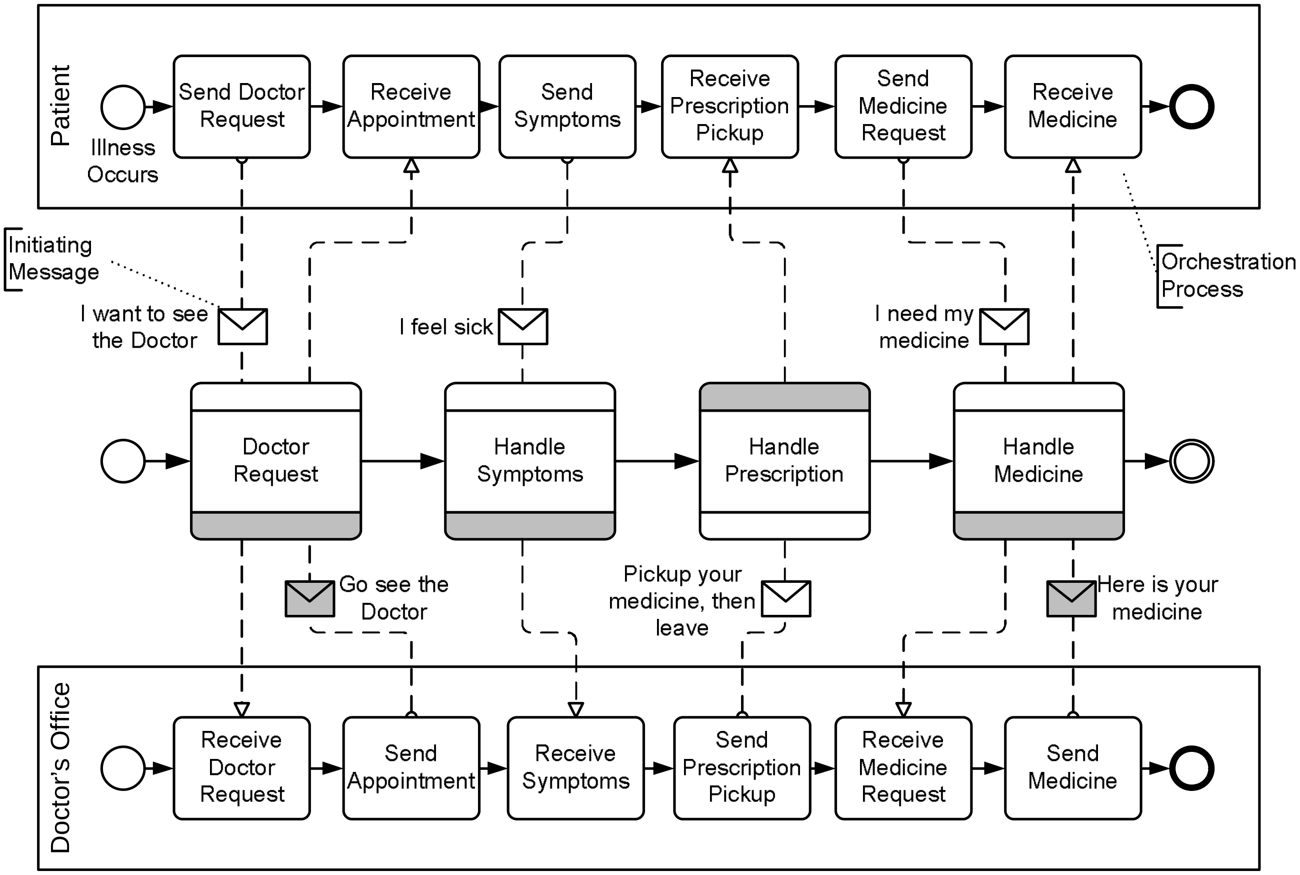


Figure 12-51 – An example of a Choreography Process combined with Pools that contain Processes

Choreography Task in Combined View

Choreography Sub-Process in Combined View

12.8. XML Schema for Choreography

Table 12-9 – Choreography XML schema

```
<xsd:element name="choreography" type="tChoreography" substitutionGroup="rootElement"/>
<xsd:complexType name="tChoreography">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="conversation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="conversationAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlowAssociation" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="isClosed" type="xsd:boolean" default="false"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 12-10 – GlobalChoreographyTask XML schema

```
<xsd:element name="globalChoreographyTask" type="tGlobalChoreographyTask"
        substitutionGroup="rootElement"/>
<xsd:complexType name="tGlobalChoreographyTask">
    <xsd:complexContent>
        <xsd:extension base="tCallableElement">
            <xsd:sequence>
                <xsd:element ref="participant" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="messageFlow" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="initiatingParticipantRef" type="xsd:QName"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 12-11 – ChoreographyActivity XML schema

```
<xsd:element name="choreographyActivity" type="tChoreographyActivity"/>
<xsd:complexType name="tChoreographyActivity" abstract="true">
    <xsd:complexContent>
        <xsd:extension base="tFlowNode">
            <xsd:sequence>
                <xsd:element name="participantRef" type="xsd:QName" minOccurs="2"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        <xsd:attribute name="initiatingParticipant" type="xsd:QName" use="required"/>
        </xsd:extension>
    </xsd:complexContent>
<xsd:complexType>
```

Table 12-12 – ChoreographyTask XML schema

```
<xsd:element name="choreographyTask" type="tChoreographyTask" substitutionGroup="flowElement"/>
<xsd:complexType name="tChoreographyTask">
    <xsd:complexContent>
        <xsd:extension base="tChoreographyActivity">
            <xsd:sequence>
                <xsd:element name="messageFlowRef" type="xsd:QName" minOccurs="1"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 12-13 – CallChoreographyActivity XML schema

```
<xsd:element name="callChoreographyActivity" type="tCallChoreographyActivity"
        substitutionGroup="flowElement"/>
<xsd:complexType name="tCallChoreographyActivity">
    <xsd:complexContent>
        <xsd:extension base="tChoreographyActivity">
            <xsd:sequence>
                <xsd:element ref="participantAssociation" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        <xsd:attribute name="calledElement" type="xsd:QName" use="optional"/>
    </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Table 12-14 – ChoreographySubProcess XML schema

```
<xsd:element name="choreographySubProcess" type="tChoreographySubProcess"
        substitutionGroup="flowElement"/>
<xsd:complexType name="tChoreographySubProcess">
    <xsd:complexContent>
        <xsd:extension base="tChoreographyActivity">
            <xsd:sequence>
                <xsd:element ref="flowElement" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element ref="artifact" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

13. BPMN Notation and Diagrams

13.1. Diagram Interchange

The goal of the Diagram Interchange (DI) metamodel is to provide a way for BPMN to persist and interchange diagrams. Having common interchange format benefits tool interoperability, which is an ever increasing demand by end users. The DI metamodel, similar to the BPMN semantic metamodel, is defined as a MOF-based metamodel and hence its instances are serialized and interchanged with XMI.

Annex B is a non-normative description of the generic diagram interchange mechanism that was applied to BPMN 2.0 in order to derive BPMN 2.0's normative diagram interchange specification. The normative BPMN 2.0 diagram interchange specification has two parts. One part is this Chapter 13 of this document. The other part is document <omg document #>, which contains the diagram interchange schema. Section 13.2 of this document defines an instance of a metamodel that is defined in Annex B..

13.2. BPMN Diagram Definition Library

This chapter describes the BPMN diagram definition (M1) library. View definitions contained in this library define views that constitute the building blocks of BPMN diagrams.

13.2.1. BPMN Diagram Definitions

The BPMN 2.0 specification defines the following types of diagrams: *Orchestration* diagram (aka Process diagram), Collaboration diagram, Conversation diagram and Choreography diagram. All of these are described in detail in Sections 9, 10, 11 and 12.

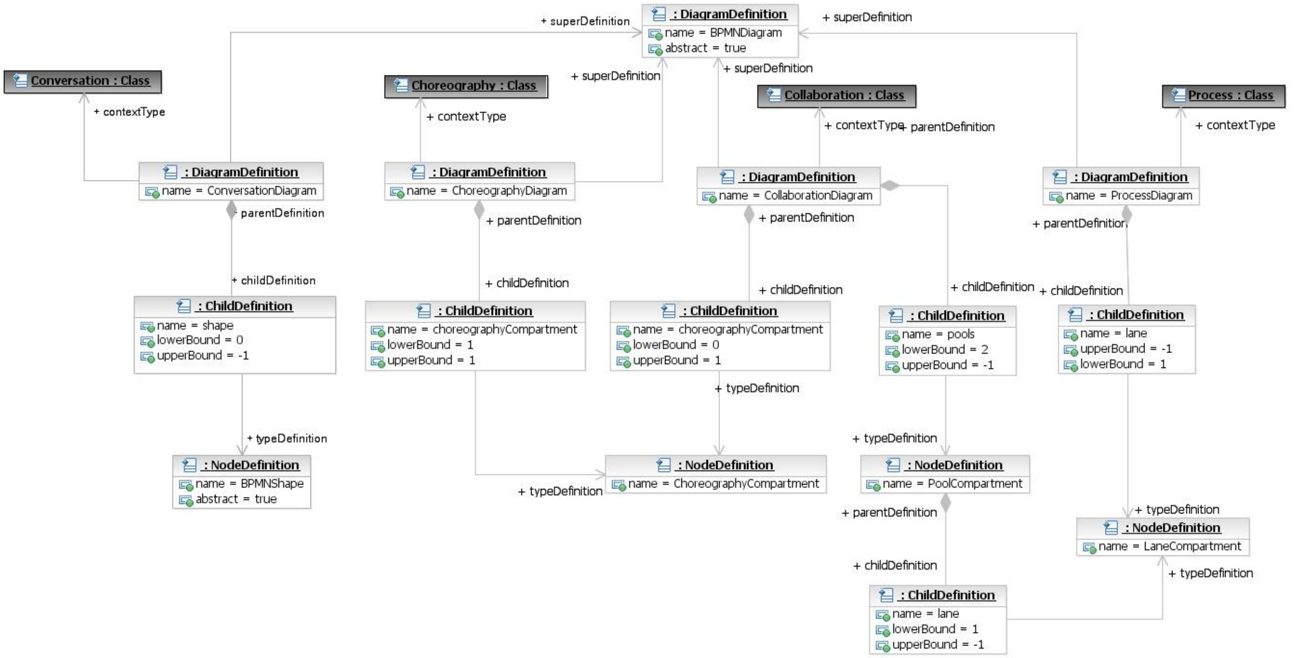


Figure 13-1 – BPMN Diagram Definitions

BPMNDiagram

BPMNDiagram is an abstract DiagramDefinition that is the super definition of all BMPN diagram definitions. BPMNDiagram has no further relationships or constraints.

ProcessDiagram

ProcessDiagram is a concrete DiagramDefinition that defines diagrams that reference a BPMN Process element as a context.

Super Definition

- BPMNDiagram

Context Type

- Process

Child Definitions

Table 13-1 – ProcessDiagram children

Attribute Name	Description/Usage
lane: LaneCompartment [1..*]	A reference to the lane compartment elements contained in the Process diagram.

CollaborationDiagram

CollaborationDiagram is a concrete DiagramDefinition that defines diagrams that reference a BPMN Collaboration element as a context.

Super Definition

- BPMNDiagram

Context Type

- Collaboration

Child Definitions

Table 13-2 – CollaborationDiagram children

Attribute Name	Description/Usage
pools: PoolCompartment [2..n*]	A reference to two or more the pool compartment elements contained in the Collaboration diagram.
choreographyCompartment: ChoreographyCompartment [0..1]	A reference to the choreography compartment element contained in the Choreography diagram.

ChoreographyDiagram

`ChoreographyDiagran` is a concrete `DiagramDefinition` that defines diagrams that reference a BPMN `Choreography` element as a context.

Super Definition

- `BPMNDiagram`

Context Type

- `Choreography`

Child Definitions

Table 13-3 – ChoreographyDiagram children

Attribute Name	Description/Usage
choreographyCompartment: ChoreographyCompartment	A reference to the choreography compartment element contained in the Choreography diagram.

ConversationDiagram

`ConversationDiagram` is a concrete `DiagramDefinition` that defines diagrams that reference a BPMN `Conversation` element as a context.

Super Definition

- `BPMNDiagram`

Context Type

- Conversation

Child Definitions

Table 13-4 – Conversation children

Attribute Name	Description/Usage
shape: BPMNShape [0..*]	The BPMN conversation shapes displayed inside the Conversation diagram.

13.2.2. BPMN Node Definition

BPMNNode is an abstract NodeDefinition that is the super definition for all BPMN Node types, like BPMN Compartments (Lanes, Pools and Choreography) and BPMN shapes (e.g. Activity).

Super Definition

- NodeDefinition

Style Definitions

Table 13-5 – BPMNNode styles

Attribute Name	Description/Usage
width: Integer = -1	Specifies the width of the compartment. Default is auto size (-1)
height: Integer = -1	Specifies the height of the compartment. Default is auto size (-1)
x: Integer = -1	Specifies the x position of the Node relative to the Node owning/containing the Node
y: Integer = -1	Specifies the y position of Node relative to the Node owning/containing the Node

13.2.3. BPMN Compartment Definitions

Every one of the diagram definitions described in the previous section contains different areas which contain and visualize certain parts of the overall BPMN diagram. These areas all called “compartments”. The BPMN 2.0

spec defines three types of these compartments which are Lanes, Pools and Choreographies. For more detailed information see Sections 9.2 and 9.4.

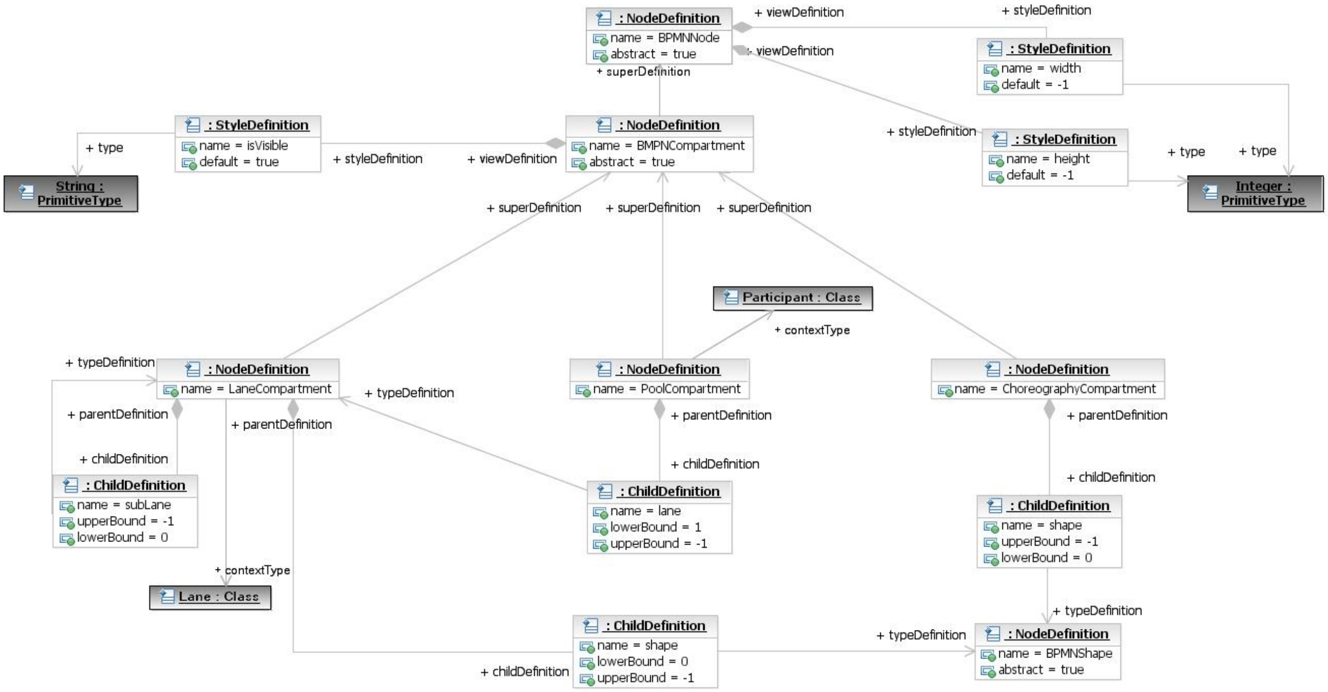


Figure 13-2 – BPMN Compartment Definitions

BPMNCompartment

BPMNCompartment is an abstract BPMNNodeDefinition that is the super definition for all BPMN compartment types, like Lanes, Pools and Choreography. A compartment visually partitions a diagram by grouping some shapes in a bounded area.

Super Definition

- BPMNNode

Style Definitions

Table 13-6 – BPMNCompartment styles

Attribute Name	Description/Usage
isVisible: Boolean = true	Whether a compartment is visible or hidden (still preserved in the model but not shown)

PoolCompartment

PoolCompartment is a concrete NodeDefinition that defines the visual element of a BPMN Pool. A Pool is the graphical representation of a *Participant* in a Collaboration (see page 146). It is also acts as a graphical container for partitioning a set of Activities from other Pools, usually in the context of B2B situations.

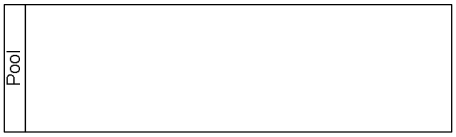


Figure 13-3 – A Pool

Super Definition

- BPMNCompartment

Child Definitions

Table 13-7 – PoolCompartment children

Attribute Name	Description/Usage
lanes: LaneCompartment [1..*]	A reference to all the lane elements contained in that pool.

LaneCompartment

LaneCompartment is a concrete NodeDefinition that defines the visual element of a BPMN lane. A Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally (see on page 149). Lanes are used to organize and categorize Activities.

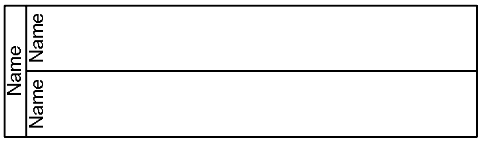


Figure 13-4 – Lanes within a Pool

Super Definition

- BPMNCompartment

Context Type

- Lane

Child Definitions

Table 13-8 – LaneCompartment children

Attribute Name	Description/Usage
shape: BPMNShape [0..*]	The BPMN shapes displayed inside the lane.
subLane: LaneCompartment [0..*]	A LaneCompartment element can optionally contain other LaneCompartments as sub lanes

ChoreographyCompartment

ChoreographyCompartment is a concrete NodeDefinition that defines the visual area displaying the Choreography Activities. The ChoreographyCompartment contains the BPMN shape elements that are displayed in that area.

Super Definition

- BPMNCompartment

Child Definitions

Table 13-9 – ChoreographyCompartment children

Attribute Name	Description/Usage
shape: BPMNShape [0..*]	The BPMN shapes displayed inside the Choreography area.

13.2.4. BPMN Connectors

The BPMN 2.0 specification defines different elements for connecting BPMN elements. These are mainly Sequence Flow connectors, Message Flow connectors, Association connectors, and Data Association connectors.

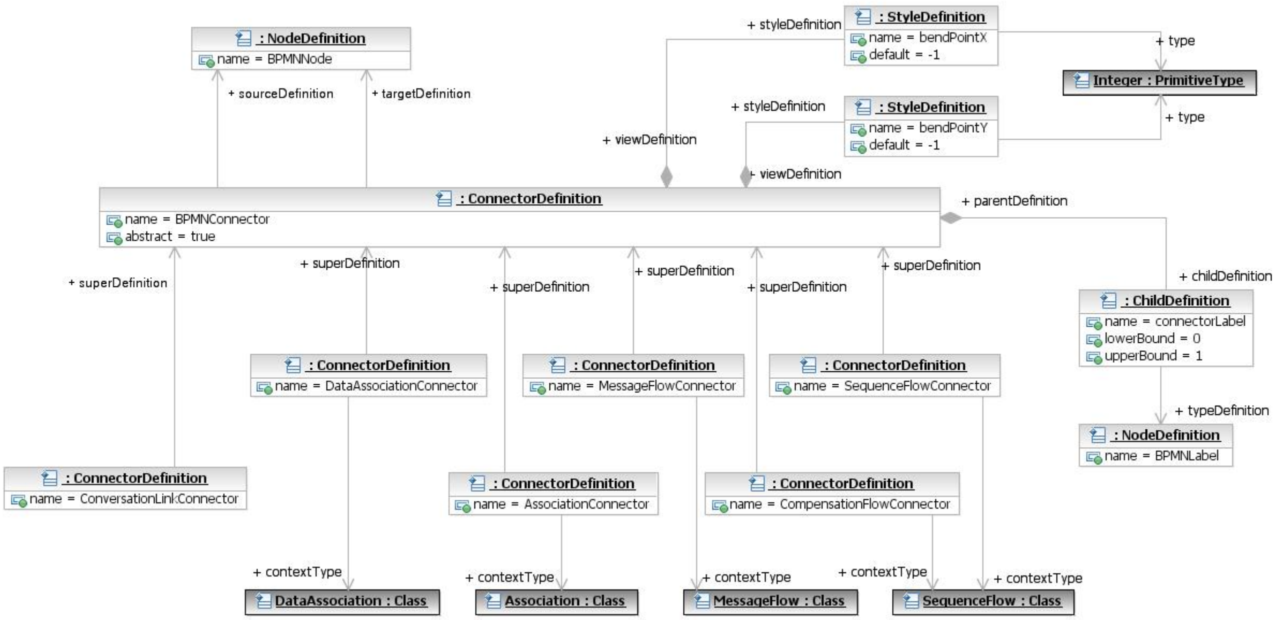


Figure 13-5 – BPMN Connectors class diagram

BPMNConnector

`BPMNConnector` is an abstract `ConnectorDefinition` that defines the visual line connecting `BPMN` nodes.

Super Definition

- `ConnectorDefinition`

Child Definitions

Table 13-10 – BPMNConnector children

Attribute Name	Description/Usage
source: BPMNNode	A reference to the Node element which is the source of the connector
target: BPMNNode	A reference to the Node element which is the target of the connector
connectorLabel: BPMNLabel [0..1]	The optional label for the Connector

SequenceFlowConnector

A Sequence Flow is used to show the order that Activities will be performed in a Process (see page 159) and in a Choreography (see page 348).

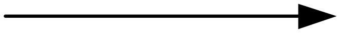


Figure 13-6 – A Sequence Flow

SequenceFlowConnector is a concrete BPMNConnector that defines the visual line representing a BPMN Sequence Flow connector.

Super Definition

- BPMNConnector

Context Type

- Sequence Flow

DataAssociationConnector

A Data Associations are used to model a data flow between Data Objects and Activities or Events.

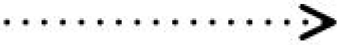


Figure 13-7 – A Data Association

DataAssociationConnector is a concrete BPMNConnector that defines the visual line representing a BPMN Data Association flow connector.

Super Definition

- BPMNConnector

Context Type

- Data Association

MessageFlowConnector

A Message Flow is used to show the flow of Messages between two *Participants* that are prepared to send and receive them (see page 119). In BPMN, two separate Pools in a Collaboration Diagram will represent the two *Participants* (e.g., business PartnerEntities or business PartnerRoles).

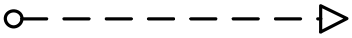


Figure 13-8 – A Message Flow

MessageFlowConnector is a concrete BPMNConnector that defines the visual line representing a BPMN Message Flow connector.

Super Definition

- BPMNConnector

Context Type

- Message Flow

AssociationConnector

An Association is used to link information and Artifacts with BPMN graphical elements (see page 88). Text Annotations (see page 93) and other Artifacts (see page 86) can be associated with the graphical elements. An arrowhead on the Association indicates a direction of flow (e.g., data), when appropriate.

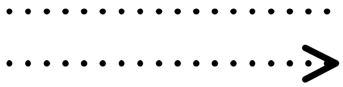


Figure 13-9 – Associations

AssociationConnector is a concrete BPMNConnector that defines the visual line representing a BPMN Association connector.

Super Definition

- BPMNConnector

Context Type

- Association

CompensationFlowConnector

Compensation Association occurs outside the *normal flow* of the Process and is based upon an Event (a Compensation Intermediate Event) that is triggered through the failure of a *transaction* or a Compensate Event (see page 270). The target of the Association must be marked as a Compensation Activity.

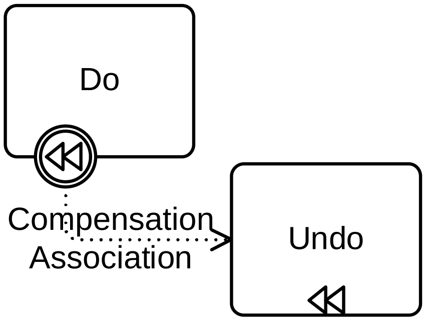


Figure 13-10 – *Compensation Flow* from a Compensation Intermediate Event

CompensationFlowConnector is a concrete BPMNConnector that defines the visual line representing a BPMN Association connector.

Super Definition

- BPMNConnector

Context Type

- Association

ConversationLinkConnector

ConversationLinkConnector is a concrete BPMNConnector that defines the visual line representing a BPMN ConversationLink connector.

Super Definition

- BPMNConnector

Context Type

ConversationLink

13.2.5. BPMN Shapes

The BPMN 2.0 specification describes different flow shapes, like Activities, Data Objects, or Text Annotations. These shape elements are contained in the corresponding BPMN diagram element and can be referenced by the BPMN-DI element LaneCompartment and ChoreographyCompartment. The position of a shape is relative to the visual owner, LaneCompartment or ChoreographyCompartment.

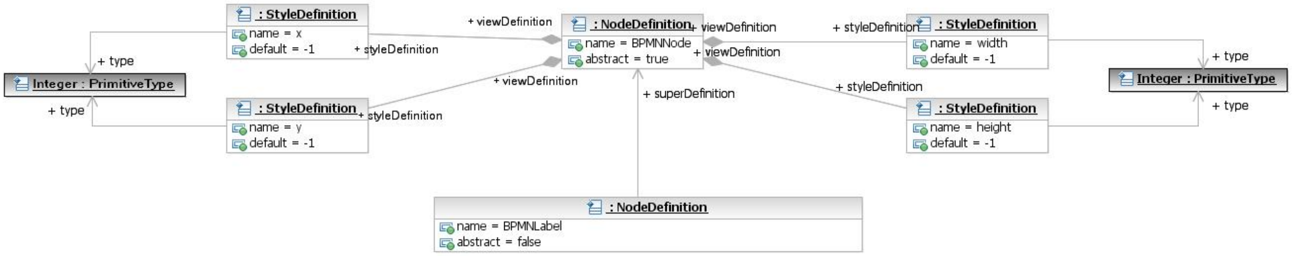


Figure 13-11 – BPMN Shapes class diagram

BPMNShape is an abstract BPMNNode.

Super Definition

- `BPMNConnector`

ActivityShape

The element `ActivityShape` represents an `Activity` contained in a `Process` model. It extends the FlowNodeShape element. The type `Activity` is determined by the referenced semantic meta model element.

An `Activity` is a generic term for work that company performs (see page 159) in `Process`. An `Activity` can be atomic or non-atomic (compound). The types of `Activities` that are a part of a `Process` model are: `Sub-Process` and `Task`, which are rounded rectangles. `Activities` are used in both standard `Processes` and in `Choreographies`.

Figure 13-12 – An Activity

`ActivityShape` is a concrete `BPMNShape`.

Super Definition

- `BPMNShape`

Context Type

- `Activity`

GatewayShape

A `Gateway` is used to control the divergence and convergence of `Sequence Flow` in a `Process` (see page 159) and in a `Choreography` (see page 375). Thus, it will determine branching, forking, merging, and joining of paths. Internal markers will indicate the type of behavior control.

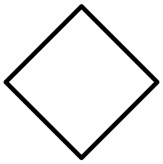


Figure 13-13 – A Gateway

`GatewayShape` is a concrete `BPMNShape`.

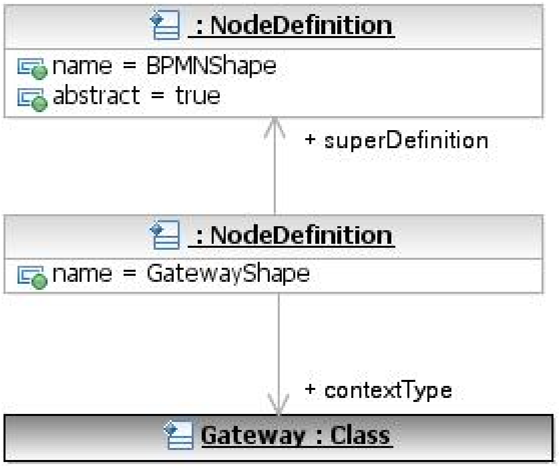


Figure 13-14 – The Gateway shape class diagram

Super Definition

- BPMNShape

Context Type

- Gateway

EventShape

An Event is something that “happens” during the course of a Process (see page 153) or a Choreography (see page 369). These Events affect the flow of the model and usually have a cause (*Trigger*) or an impact (*Result*). Events are circles with open centers to allow internal markers to differentiate different *Triggers* or *Results*. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End.

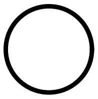


Figure 13-15 – An Event

EventShape is a concrete BPMNShape

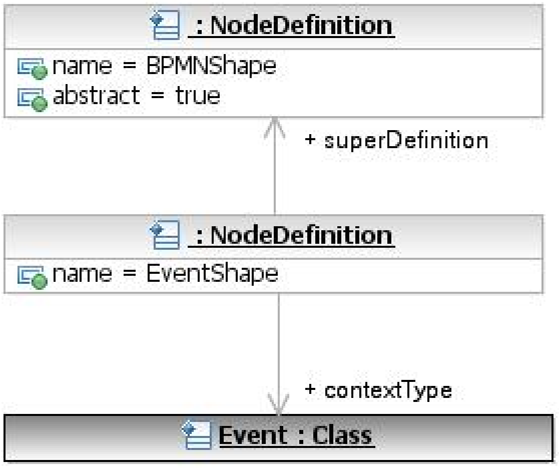


Figure 13-16 – The Event shape class diagram

Super Definition

- BPMNShape

Context Type

- Event

DataObjectShape

Data Objects provide information about what Activities require to be performed and/or what they produce (see page 213), Data Objects can represent a singular object or a collection of objects.

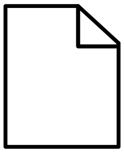


Figure 13-17 – A Data Object

Collection

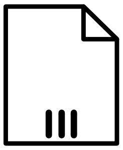


Figure 13-18 – A *Collection* Data Object

DataObjectShape is a concrete BPMNShape.

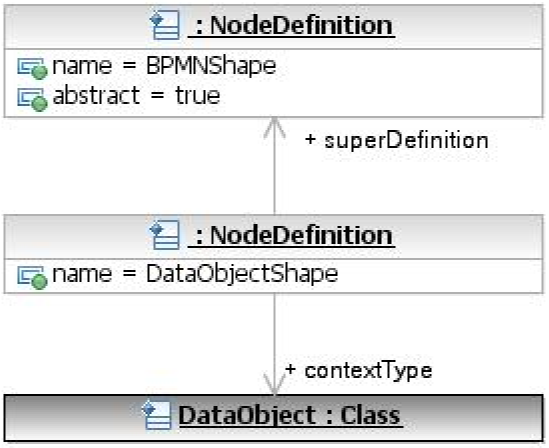


Figure 13-19 – The Data Object shape class diagram

Super Definition

- BPMNShape

Context Type

- Data Object

DataStoreShape

Data Stores provide a source for Activities to retrieve data or store data (see page 215),

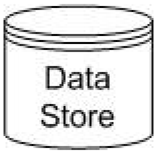


Figure 13-20 – A Data Object

DataStoreShape is a concrete BPMNShape.

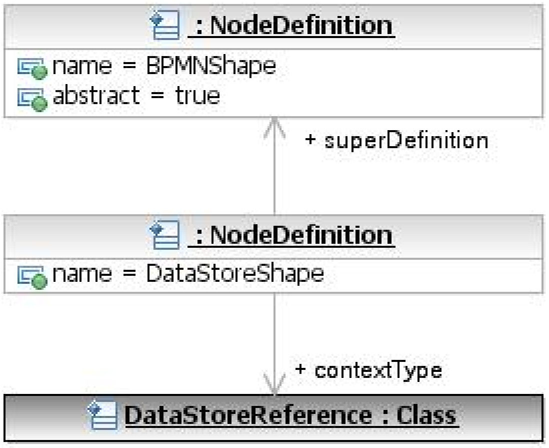


Figure 13-21 – The Data Store shape class diagram

Super Definition

- `BPMNShape`

Context Type

- `DataStoreReference`

DataInputShape

`Data Input` provide information about what `Processes` require to be performed. Data Inputs are part of the Input Output Specification.

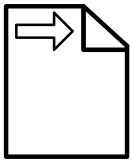


Figure 13-22 – A Data Input

Collection

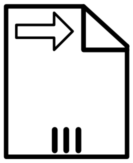


Figure 13-23 – A *Collection* Data Input

`DataInputShape` is a concrete `BPMNShape`

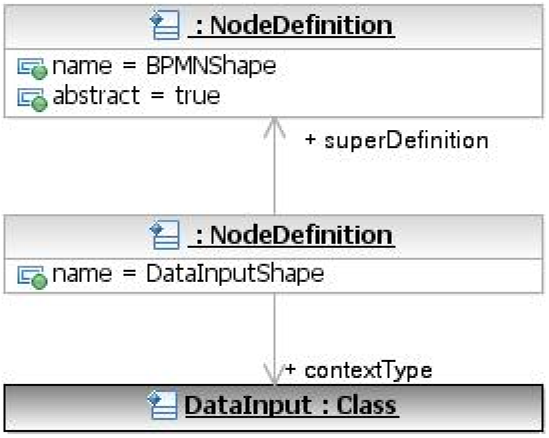


Figure 13-24 – The Data Input shape class diagram

Super Definition

- `BPMNShape`

Context Type

- `DataInput`

DataOutputShape

`Data Outputs` provide information about what a `Process` produce as output available to the caller of the `Process`.

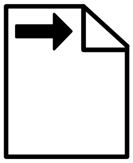


Figure 13-25 – A Data Output

Collection

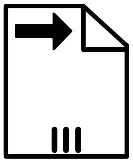


Figure 13-26 – A *Collection* Data Output

`DataOutputShape` is a concrete `BPMNShape`.

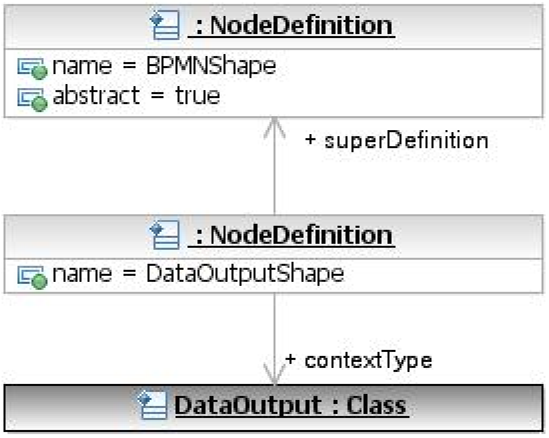


Figure 13-27 – The Data Output shape class diagram

Super Definition

- `BPMNShape`

Context Type

- `DataOutput`

MessageShape

A `Message` is used to depict the contents of a communication between two *Participants* (as defined by a business `PartnerRole` or a business `PartnerEntity`—see page 112).

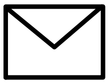


Figure 13-28 – A Message

`MessageShape` is a concrete `BPMNShape`.

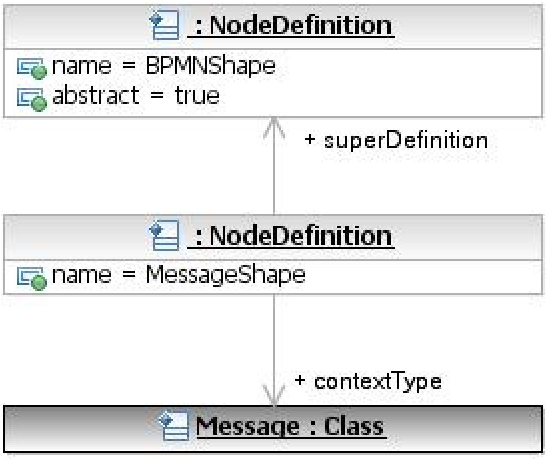


Figure 13-29 – The Message shape class diagram

Super Definition

- BPMNShape

Context Type

- Message

ChoreographyActivityShape

A Choreography Task is an atomic Activity in a Choreography (see page 350). It represents a set of one (1) or more Message exchanges. Each Choreography Task involves two (2) or more *Participants*. The name of the Choreography Task and each of the *Participants* are all displayed in the different bands that make up the shape's graphical notation. There are two (2) more Participant Bands and one Task Name Band.

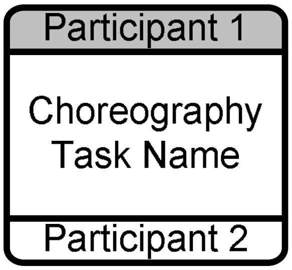


Figure 13-30 – A Choreography Task

ChoreographyShape is a concrete BPMNShape.

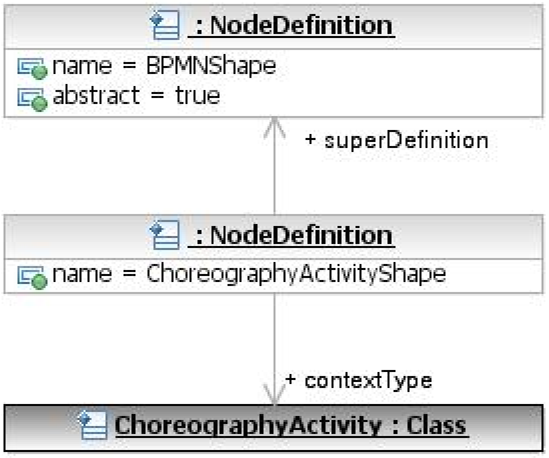


Figure 13-31 – The Choreography Task shape class diagram

Super Definition

- BPMNShape

Context Type

- Choreography Activity

GroupShape

A Group is a grouping of Activities that are within the same *Category* (see page 89). This type of grouping does not affect the Sequence Flow of the Activities within the Group. The *Category* name appears on the diagram as the Group label. *Categories* can be used for documentation or analysis purposes. Groups are one way in which *Categories* of objects can be visually displayed on the diagram.

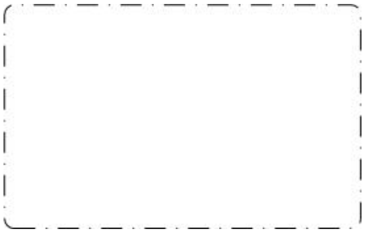


Figure 13-32 – A Group

GroupShape is a concrete BPMNShape.

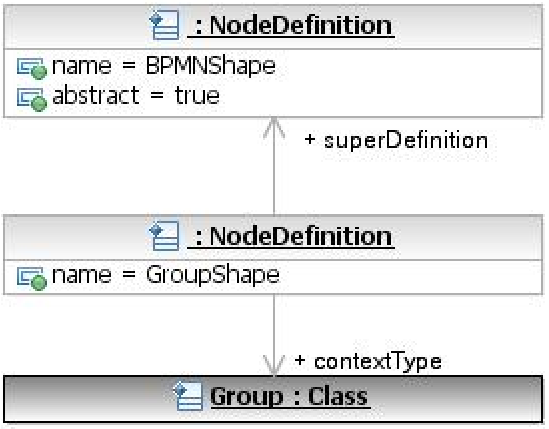


Figure 13-33 – The Group shape class diagram

Super Definition

- BPMNShape

Context Type

- Group

TextAnnotationShape

Text Annotations are a mechanism for a modeler to provide additional text information for the reader of a BPMN Diagram (see page 92).

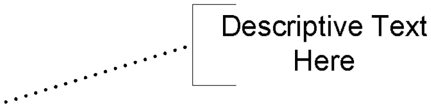


Figure 13-34 – A Text Annotation

TextAnnotationShape is a concrete BPMNShape.

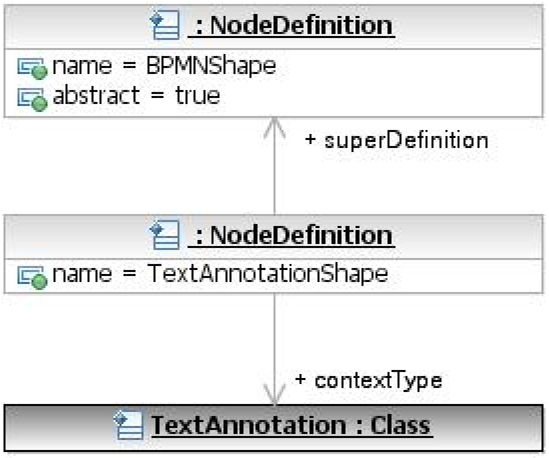


Figure 13-35 – The Text Annotation shape class diagram

Super Definition

- BPMNShape

Context Type

- Text Annotation

EmbeddedSubProcessShape

A Sub-Process is a compound Activity that is included within a Process (see page 176). It is compound in that it can be broken down into a finer level of detail (a Process through a set of sub-Activities).

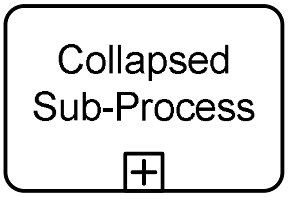


Figure 13-36 – A Sub-Process object (collapsed)

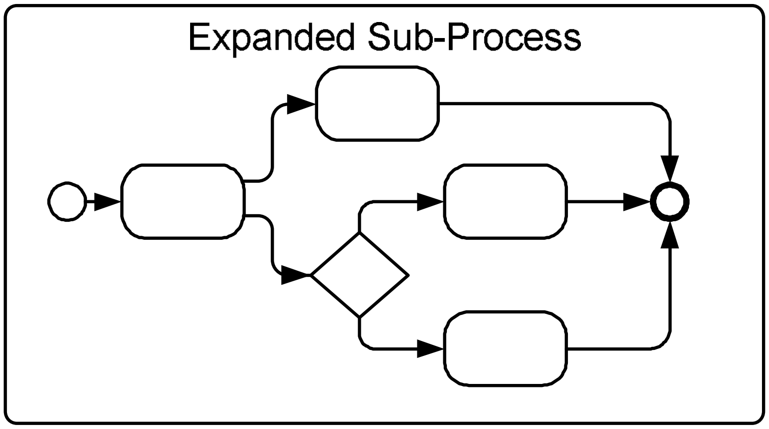


Figure 13-37 – A Sub-Process object (expanded)

`EmbeddedSubProcessShape` is a concrete `BPMNShape`.

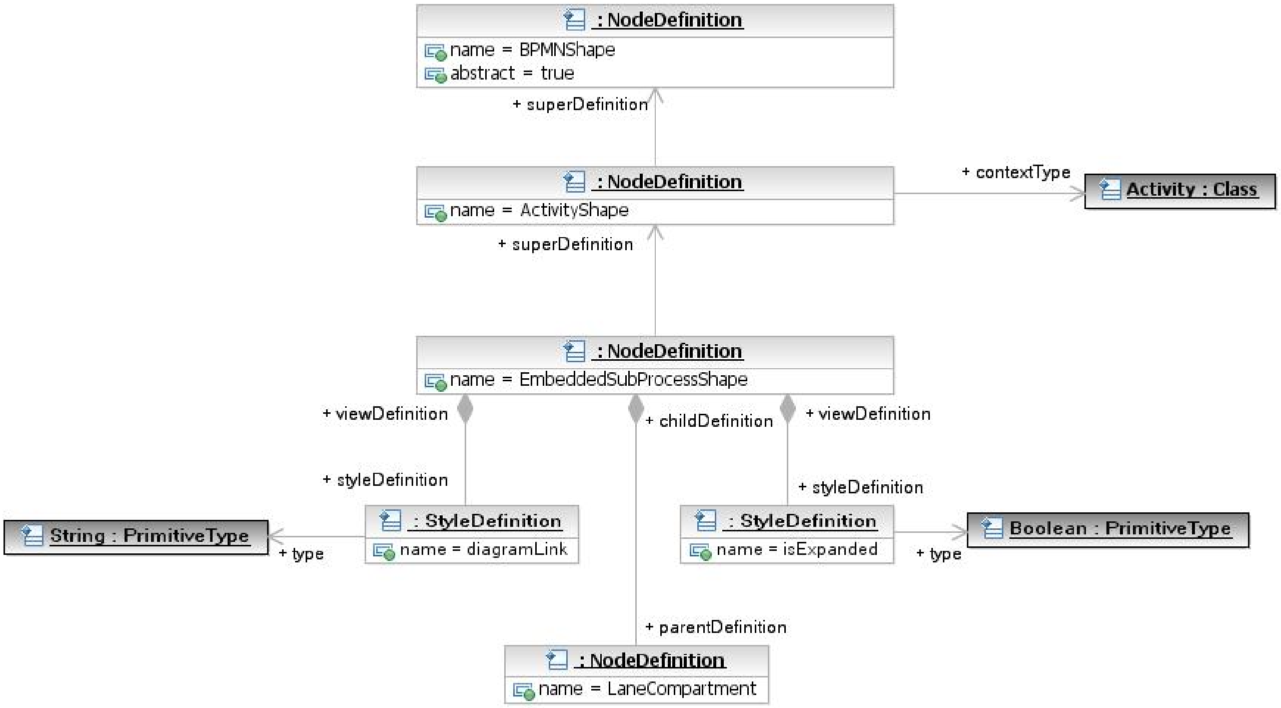


Figure 13-38 – The Sub-Process shape class diagram

Super Definition

- `ActivityShape`

Context Type

- `Activity`

Style Definitions

Table 13-11 – EmbeddedSubProcessShape styles

Attribute Name	Description/Usage
diagramLink: String	A reference to another BPMN diagram defining the details of the Subprocess.
isExpanded: Boolean	Indicated whether the Subprocess is expanded or collapsed.

Child Definitions

Table 13-12 – EmbeddedSubProcessShape children

Attribute Name	Description/Usage
lane: LaneCompartment[0..n]	A reference to all the lane elements contained in that Subprocess. In case the SubprocessShape links to another diagram containing the subprocess details, a Lane doesn't have to specified.

CalledSubProcessShape

A Sub-Process is a compound Activity that is included within a Process (see page 176). It is compound in that it can be broken down into a finer level of detail (a Process through a set of sub-Activities).

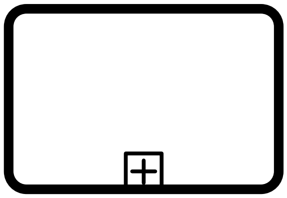


Figure 13-39 – A Call Activity object calling a Process (Collapsed)

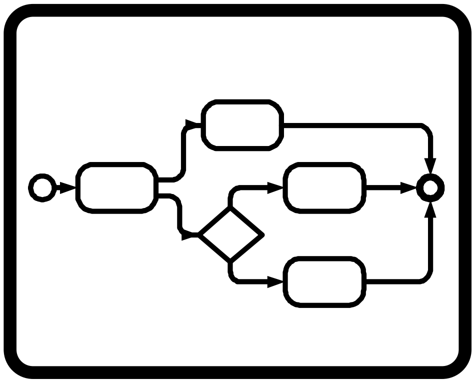


Figure 13-40 – A Call Activity object calling a Process (Expanded)

`CalledSubProcessShape` is a concrete `BPMNShape`.

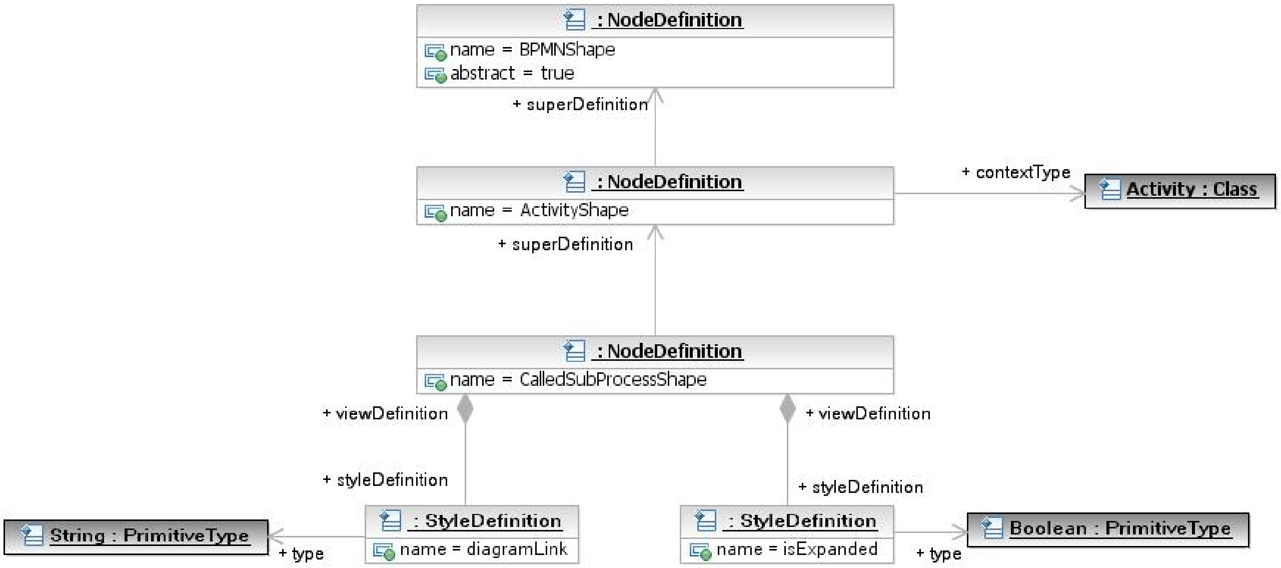


Figure 13-41 – The Call Activity shape class diagram

Super Definition

- `ActivityShape`

Context Type

- `Activity`

Style Definitions

Table 13-13 – CalledSubProcessShape styles

Attribute Name	Description/Usage
diagramLink: String	A reference to another BPMN diagram defining the details of the Subprocess.
isExpanded: Boolean	Indicated whether the Subprocess is expanded or collapsed.

CommunicationShape

`CommunicationShape` is a concrete `BPMNShape`.

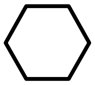


Figure 13-42 – A Communication shape

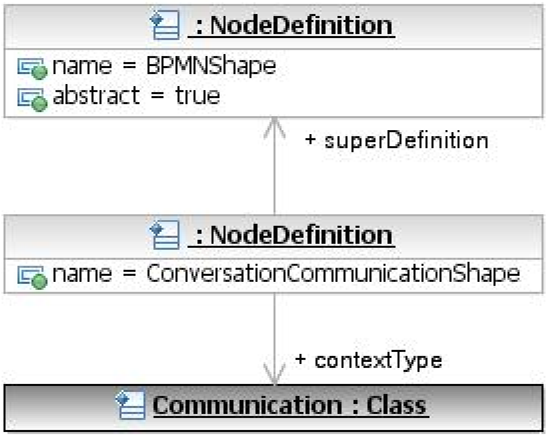


Figure 13-43 – The ConversationCommunication shape class diagram

Super Definition

- `BPMNShape`

Context Type

- `Communication`

SubConversationShape

`SubConversationShape` is a concrete `BPMNShape`.

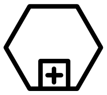


Figure 13-44 – A Sub-Conversation shape

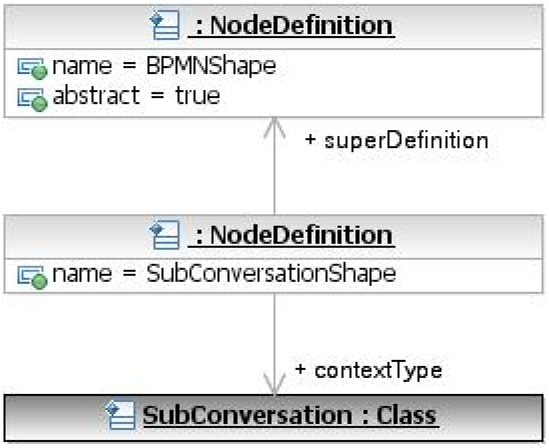


Figure 13-45 – The SubConversation shape class diagram

Super Definition

- `BPMNShape`

Context Type

- `SubConversation`

CallConversationShape

`CallConversationShape` is a concrete `BPMNShape`.

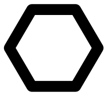


Figure 13-46 – A Call Conversation object calling a GlobalCommunication

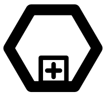


Figure 13-47 – A Call Conversation object calling a Conversation

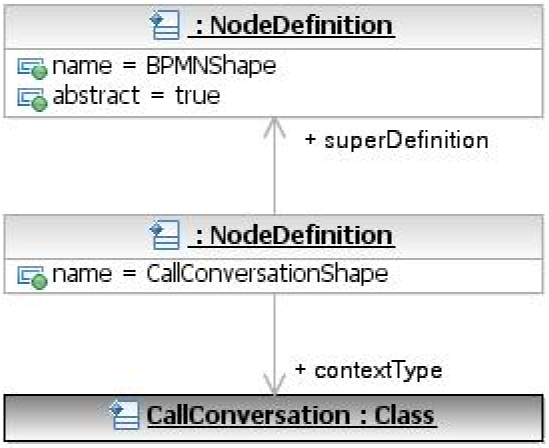


Figure 13-48 – The CallConversation shape class diagram

Super Definition

- BPMNShape

Context Type

- CallConversation

13.2.6. BPMN Label

BPMNLabel is a concrete BPMNNode that defines a label for BPMN connectors, like SequenceFlowConnector.

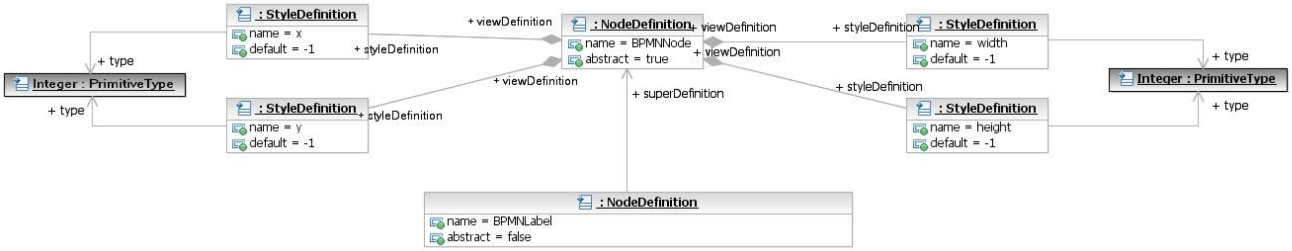


Figure 13-49 – The BPMN Label class diagram

Super Definition

- BPMNNode

14. BPMN Execution Semantics

Note: The content of this chapter is required for BPMN **Process Execution Conformance** or for BPMN **Complete Conformance**. However, this chapter is not required for BPMN **Process Modeling Conformance**, BPMN **Choreography Conformance**, or BPMN **BPEL Process Execution Conformance**. For more information about BPMN conformance types, see page 28.

This section defines the execution semantics for orchestrations in BPMN 2.0. The purpose of this execution semantics is to describe a clear and precise understanding of the operation of the elements. However, for some elements only conceptual model is provided which does not specify details needed to execute them on an engine. These elements are called non-operational. Implementations MAY extend the semantics of non-operational elements to make them executable, but this is considered to be an optional extension to BPMN. Non-operational elements MAY be ignored by implementations conforming to BPMN **Process Execution Conformance** type. The following elements are non-operational:

- Manual Task
- Abstract Task
- DataState
- IORules
- Ad-Hoc Process
- ItemDefinitions with an itemKind of Physical
- the inputSetWithWhileExecuting attribute of DataInput
- the outputSetWithWhileExecuting attribute of DataOutput
- the isClosed attribute of Process
- the isImmediate attribute of Sequence Flow

The execution semantics are described informally (textually), and this based on prior research involving the formalization of execution semantics using mathematical formalisms.

This section provides the execution semantics of elements through the following structure:

- A description of the operational semantics of the element,
- Exception issues for the element where relevant,
- List of workflow patterns[3] supported by the element where relevant.

[3] http://www.workflowpatterns.com/patterns/control/index.php

14.1. Process Instantiation and Termination

A Process is instantiated when one of its Start Events occurs. Each occurrence of a Start Event creates a new Process *Instance* unless the Start Event participates in a Conversation that includes other Start Events. In that case, a new Process *instance* is only created if none already exists for the specific Conversation (identified through its associated correlation information) of the Event occurrence. Subsequent Start Events that share the same correlation information as a Start Event that created a Process *instance* are routed to that Process *instance*. Note that a *global* Process must neither have any empty Start Event nor any Gateway or Activity without *incoming* Sequence Flow. An exception is the Event Gateway.

A Process can also be started via an Event-Based Gateway that has no *incoming* Sequence Flow and its Instantiate flag is *true*. If the Event-Based Gateway is *exclusive*, the first matching Event will create a new *instance* of the Process. The Process then does not wait for the other Events originating from the same Event-Based Gateway (see also semantics of the Event-Based Exclusive Gateway on page 437). If the Event-Based Gateway is *parallel*, also the first matching Event creates a new Process *instance*. However, the Process then waits for the other Events to arrive. As stated above, those Events must have the same correlation information as the Event that arrived first. A Process *instance* completes only if all Events that succeed a Parallel Event-Based Gateway have occurred.

To specify that the instantiation of a Process waits for multiple Start Events to happen, a Multiple Parallel Start Event can be used.

Note that two Start Events are alternative, A Process *instance* triggered by one (1) of the Start Events does not wait for an alternative Start Event to occur. Note that there may be multiple instantiating Parallel Event-Based Gateways. This allows the modeler to express that either all the Events after the first Gateway occur or all the Events after the second Gateway and so forth.

Each Start Event that occurs creates a *token* on its *outgoing* Sequence Flow, which is followed as described by the semantics of the other Process elements.

A Process *instance* is completed, if and only if the following three conditions hold:

- If the *instance* was created through an instantiating Parallel Gateway, then all subsequent Events (of that Gateway) must have occurred.
- There is no *token* remaining within the Process *instance*.
- No Activity of the Process is still active.

For a Process *instance* to become completed, all *Tokens* in that *instance* must reach an end node, i.e., a node without *outgoing* Sequence Flow. A *Token* reaching an End Event triggers the behavior associated with the Event type is, e.g., the associated Message is sent for a Message End Event, the associated *Signal* is sent for a Signal End Event, and so on. If a *Token* reaches a Terminate End Event, the entire Process is abnormally terminated.

14.2. Activities

This section specifies the semantics of Activities. First the semantics that is common to all Activities is described. Subsequently the semantics of special types of Activities is described.

14.2.1. Sequence Flow Considerations

The nature and behavior of Sequence Flow is described in Section 8.3.17. But there are special considerations relative to Sequence Flow when applied to Activities. An Activity that is the target of multiple Sequence Flow participates in "uncontrolled flow."

To facilitate the definition of Sequence Flow (and other Process elements) behavior, we employ the concept of a *token* that will traverse the Sequence Flow and pass through the elements in the Process. A *token* is a theoretical concept that is used as an aid to define the behavior of a Process that is being performed. The behavior of Process elements can be defined by describing how they interact with a *token* as it "traverses" the structure of the Process. However, modeling and execution tools that implement BPMN are not required to implement any form of *token*.

Uncontrolled flow means that, for each *Token* arriving on any *incoming* Sequence Flow into the Activity, the Task will be enabled independently of the arrival of *Tokens* on other *incoming* Sequence Flow. The presence of multiple *incoming* Sequence Flow behaves as an exclusive gateway. If the flow of *Tokens* into the Task needs to be 'controlled', then Gateways (other than Exclusive) should be explicitly included in the Process flow prior to the Task to fully eliminate semantic ambiguities.

If an Activity has no *incoming* Sequence Flow, and there are no Start Events in the containing Process or Sub-Process, the Activity will be instantiated when the containing Process or Sub-Process is instantiated. Exceptions to this are Event Sub-Processes and Activities marked as Compensation Activities, as they have specialized instantiation behavior.

Activities can also be source of Sequence Flow. If an Activity has multiple *outgoing* Sequence Flow, all of them will receive a *token* when the Activity transitions to the *Completed* state. Semantics for *token* propagation for other termination states is defined below. Thus, multiple *outgoing* Sequence Flow behaves as a parallel split. Multiple *outgoing* Sequence Flow with conditions behaves as an inclusive split. A mix of multiple *outgoing* Sequence Flow with and without conditions is considered as a combination of a parallel and an inclusive split as shown in the following figure.

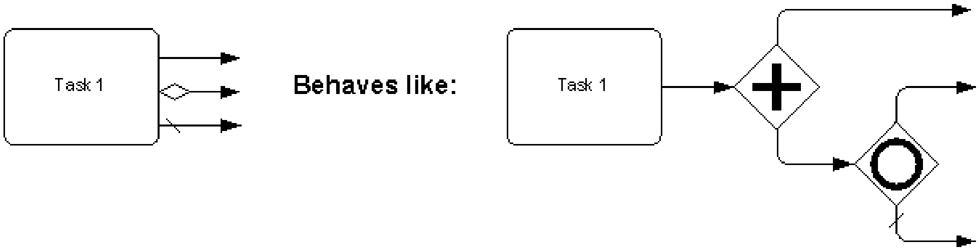


Figure 14-1 – Behavior of multiple outgoing sequence flow of an Activity

If the Activity has no *outgoing* Sequence Flow and there are no End Events in the containing Process or Sub-Process, the Activity will terminate and termination semantics for the container applied.

Token movement across a Sequence Flow does not have any timing constraints. A *token* might take a long or short time to move across the Sequence Flow. If the isImmediate attribute of a Sequence Flow has a value of *false*, or has no value and is taken to mean *false*, then Activities not in the model MAY be executed while the *token* is moving along the Sequence Flow. If the isImmediate attribute of a Sequence Flow has a value of *true*, or has no value and is taken to mean *true*, then Activities not in the model MAY NOT be executed while the *token* is moving along the Sequence Flow.

14.2.2. Activity

An Activity is a Process step that can be atomic (Tasks) or decomposable (Sub-Processes) and is executed by either a system (automated) or humans (manual). All Activities share common attributes and behavior such as states and state transitions. An Activity, regardless of type, has lifecycle generally characterizing its operational semantics. The lifecycle, described as a UML state diagram in Figure 14-2, entails states and transitions between the states.

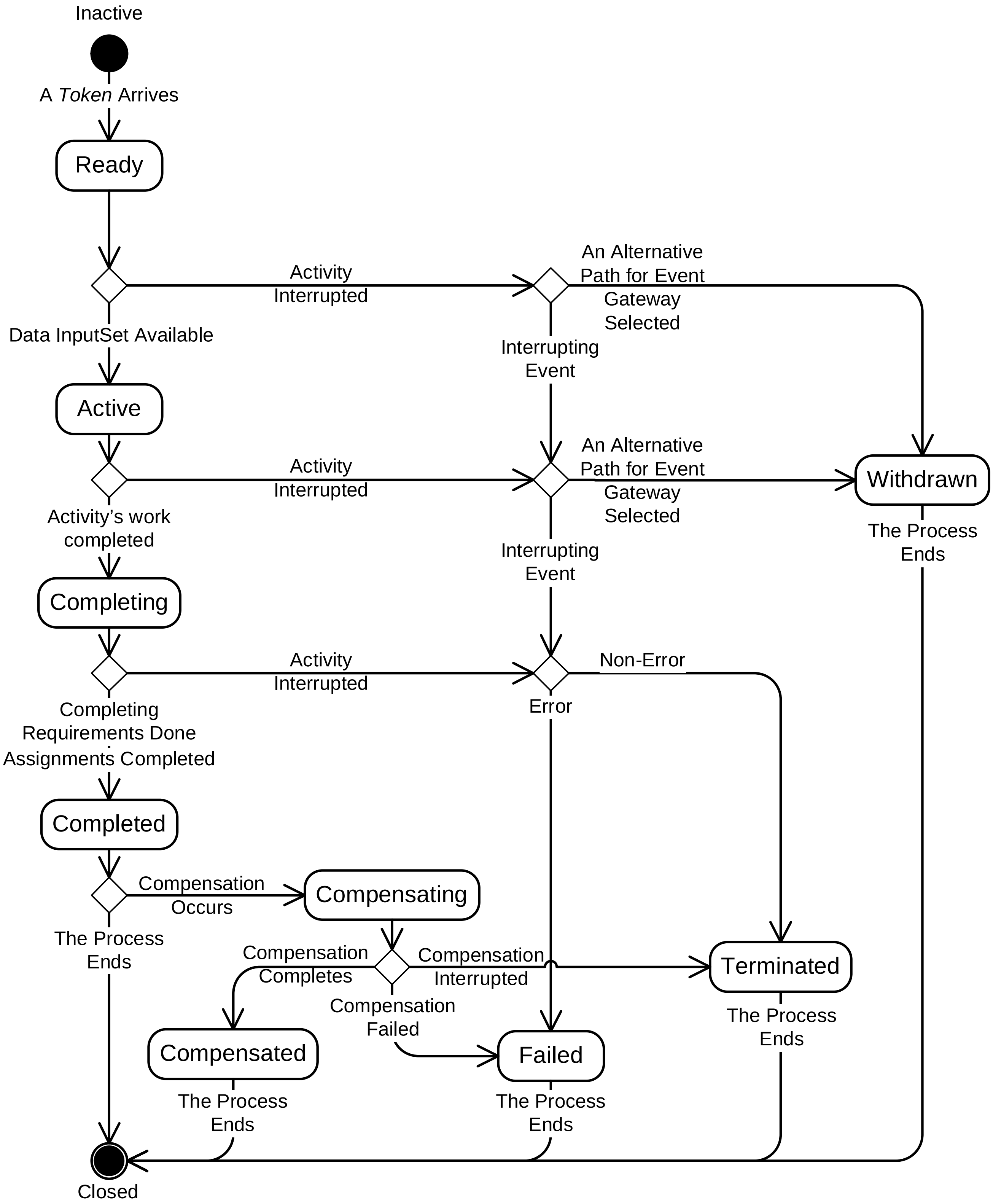


Figure 14-2 – The Lifecycle of a BPMN Activity

The lifecycle of an Activity is described as follows:

- An Activity is *Ready* for execution if the required number of *Tokens* is available to activate the Activity. The required number of *Tokens* (one or more) is indicated by the attribute StartQuantity. If the Activity has more than one *Incoming* Sequence Flow, there is an implied Exclusive Gateway that defines the behavior.
- When some data InputSet becomes available, the Activity changes from *Ready* to the *Active* state. The availability of a data InputSet is evaluated as follows. The data InputSets are evaluated in order. For each InputSet, the data inputs are filled with data coming from the elements of the context such as Data Objects or Properties by triggering the input Data Associations. An InputSet is *available* if each of its *required* data inputs is available. A data input is *required* by a data InputSet if it is not optional in that InputSet. If an InputSet is available, it is used to start the Activity. Further InputSets are not evaluated. If an InputSet is not available, the next InputSet is evaluated. The Activity waits until one InputSet becomes available.
- An Activity, if *Ready or Active*, can be *Withdrawn* from being able to complete in the context of a race condition. This situation occurs for Tasks that are attached after an Event-Based Exclusive Gateway. The first element (Task or Event) that completes causes all other Tasks to be withdrawn.
- If an Activity fails during execution, it changes from the state *Active* to *Failed*.
  - If a fault happens in the environment of the Activity, termination of the Activity is triggered, causing the Activity to go into the state *Terminated*.
- If an Activity's execution ends without anomalies, the Activity's state changes to *Completing*. This intermediate state caters for processing steps prior to completion of the Activity. An example of where this is useful is when non-interrupting *Event Handlers* (proposed for BPMN 2.0) are attached to an Activity. They need to complete before the Activity to which it is attached can complete. The state *Completing* of the main Activity indicates that the execution of the main Activity has been completed, however, the main Activity is not allowed to be in the state *Completed*, as it still has to wait for all non-interrupting *Event Handlers* to complete. The state Completing does not allow further processing steps, otherwise allowed during the execution of the Activity. For example, new attached non-interrupting *Event Handlers* may be created as long as the main Activity is in state *Active*. However, once in the state *Completing*, running handlers should be completed with no possibility to create new ones.
- After all completion dependencies have been fulfilled, the state of the Activity changes to *Completed*. The *outgoing* Sequence Flow becomes active and a number of *Tokens*, indicated by the attribute CompletionQuantity, is placed on it. If there is more than one (1) outbound Sequence Flow for an Activity, it behaves like an implicit Parallel Gateway. Upon completion, also a data OutputSet of the Activity is selected as follows. All OutputSets are checked for availability in order. An OutputSet is available if all its required data outputs are available. A data output is required by an OutputSet if it is not optional in that OutputSet. If the data OutputSet is available, data is pushed into the context of the Activity by triggering the output Data Associations of all its data outputs. Further OutputSets are not evaluated. If the data OutputSet is not available, the next data OutputSet is checked. If no OutputSet is available, a runtime exception is thrown. If the Activity has an associated IORule, the chosen OutputSet is checked against that IORule, i.e., it is checked whether the InputSet that was used in starting the Activity *instance* is together with the chosen OutputSet compliant with the IORule. If not, a runtime exception is thrown.
- Only completed Activities could, in principle, be compensated, however, the Activity can end in state *Completed*, as *compensation* might not be triggered or there might be no *compensation handler* specified. If the *compensation handler* is invoked, the Activity changes to state *Compensating* until

either *compensation* finishes successfully (state *Compensated*), an exceptions occurs (state *Failed*) or controlled or uncontrolled termination is triggered (state *Terminated*).

14.2.3. Task

Task execution and completion for the different Task types are as follows:

- Service Task: Upon instantiation, the associated service is invoked. On completion of the service, the Service Task completes. If the invoked service returns a fault, that fault is treated as interrupting error, and the Activity fails.
- Send Task: Upon instantiation, the associated Message is sent and the Send Task completes.
- Receive Task: Upon instantiation, the Receive Task begins waiting for the associated Message. When the Message arrives, the Receive Task completes.
- User Task: Upon instantiation, the User Task is distributed to the assigned person or group of people. When the work has been done, the User Task completes.
- Manual Task: Upon instantiation, the manual task is distributed to the assigned person or group of people. When the work has been done, the Manual Task completes. This is a conceptual model only; a Manual Task is never actually executed by an IT system.
- Business Rule Task: Upon instantiation, the associated business rule is called. On completion of the business rule, the Business Rule Task completes.
- Script Task: Upon instantiation, the associated script is invoked. On completion of the script, the Script Task completes.
- Abstract Task: Upon instantiation, the Abstract Task completes. This is a conceptual model only; an Abstract Task is never actually executed by an IT system.

14.2.4. Sub-Process/Call Activity

A Sub-Process is an Activity which encapsulates a Process which is in turn modeled by Activities, Gateways, Events and Sequence Flow. Once a Sub-Process is instantiated, its elements behave as in a normal Process. The instantiation and completion of a Sub-Process is defined as follows.

- A Sub-Process is instantiated when it is reached by a Sequence Flow *token*. The Sub-Process has either a unique empty Start Event, which gets a *token* upon instantiation, or it has no Start Event but Activities and Gateways without *incoming* Sequence Flow. In the latter case all such Activities and Gateways get a *token*. A Sub-Process must not have any non-empty Start Events.
- If the Sub-Process does not have incoming Sequence Flow but Start Events that are target of Sequence Flow from outside the Sub-Process, the Sub-Process is instantiated when one of these Start Events is reached by a *token*. Multiple such Start Events are alternative, i.e., each such Start Event that is reached by a *token* generates a new *instance*.
- A Sub-Process *instance* completes when there are no more *tokens* in the Sub-Process and none of its Activities is still active.
- If a “terminate” End Event is reached, the Sub-Process is abnormally terminated. For a “cancel” End Event, the Sub-Process is abnormally terminated and the associated *Transaction* is aborted.

Control leaves the Sub-Process through a cancel intermediate boundary Event. For all other End Events, the behavior associated with the Event type is performed, e.g., the associated Message is sent for a Message End Event, the associated signal is sent for a signal End Event, and so on.

- If a global Process is called through a Call Activity, then the Call Activity has the same instantiation and termination semantics as a Sub-Process. However, in contrast to a Sub-Process, the global Process that is called may also have non-empty Start Events. These non-empty Start Events are alternative to the empty Start Event and hence they are ignored when the Process is called from another Process.

14.2.5. Ad-Hoc Sub-Process

An Ad-Hoc Sub-Process or Process contains a number of embedded inner Activities and is intended to be executed with a more flexible ordering compared to the typical routing of Processes. Unlike regular Processes, it does not contain a complete, structured BPMN diagram description—i.e., from Start Event to End Event. Instead the Ad-Hoc Sub-Process contains only Activities, Sequence Flow, Gateways and Intermediate Events. An Ad-Hoc Sub-Process may also contain Data Objects and Data Associations. The Activities within the Ad-Hoc Sub-Process are not required to have *incoming* and *outgoing* Sequence Flow. However, it is possible to specify Sequence Flow between some of the contained Activities. When used, Sequence Flow will provide the same ordering constraints as in a regular Process. To have any meaning, Intermediate Events will have an *outgoing* Sequence Flow and they can be triggered multiple times while the Ad-Hoc Sub-Process is active.

The contained Activities are executed sequentially or in parallel, they can be executed multiple times in an order that is only constrained through the specified Sequence Flow, Gateways, and data connections.

Operational semantics:

- At any point in time, a subset of the embedded Activities is *enabled*. Initially, all Activities without *incoming* Sequence Flow are enabled. One of the enabled Activities is selected for execution. This is not done by the implementation but usually by a *Human Performer*. If the ordering attribute is set to sequential, another enabled Activity can be selected for execution only if the previous one has terminated. If the ordering attribute is set to parallel, another enabled Activity can be selected for execution at any time. This implies the possibility of the multiple parallel *instances* of the same inner Activity.
- After each completion of an inner Activity, a condition specified through the completionCondition attribute is evaluated:
    - If *false*, the set of enabled inner Activities is updated and new Activities can be selected for execution.
    - If *true*, the Ad-Hoc Sub-Process completes without executing further inner Activities. In case the ordering attribute is set to parallel and the attribute cancelRemainingInstances is *true*, running *instances* of inner Activities are canceled. If cancelRemainingInstances is set to *false*, the Ad-Hoc Sub-Process completes after all remaining inner *instances* have completed or terminated.
- When an inner Activity with *outgoing* Sequence Flow completes, a number of *tokens* are produced on its *outgoing* Sequence Flow. This number is specified through its attribute completionQuantity. The resulting state may contain also other *tokens* on *incoming* Sequence Flow of either Activities, converging Parallel or Complex Gateways, or an Intermediate

Event. Then all *tokens* are propagated as far as possible, i.e., all activated Gateways are executed until no Gateway and Intermediate Event is activated anymore. Consequently, a state is obtained where each *token* is on an *incoming* Sequence Flow of either an inner Activity, a converging Parallel or Complex Gateway or an Intermediate Event. An inner Activity is now enabled if it has either no *incoming* Sequence Flow or there are sufficiently many *tokens* on its *incoming* Sequence Flow (as specified through startQuantity).

Workflow patterns: WCP-17 Interleaved parallel routing.

14.2.6. Loop Activity

The Loop Activity is a type of Activity that acts as a wrapper for an inner Activity that can be executed multiple times in sequence.

Operational semantics: Attributes can be set to determine the behavior. The Loop Activity executes the inner Activity as long as the loopCondition evaluates to *true*. A testBefore attribute is set to decide when the loopCondition should be evaluated: either *before* the Activity is executed or *after*, corresponding to a pre- and post-tested *loop* respectively. A loopMaximum attribute can be set to specify a maximal number of iterations. If it is not set, the number is unbounded.

Workflow Patterns Support: WCP-21 Structured Loop.

14.2.7. Multiple Instances Activity

The *multi-instance* (MI) Activity is a type of Activity that acts as a wrapper for an Activity which has multiple *instances* spawned in parallel or sequentially.

Operational semantics: The MI specific attributes are used to configure specific behavior. The attribute isSequential determines whether *instances* are generated sequentially (*true*) or in parallel (*false*). The number of *instances* to be generated is either specified by the integer-valued expression loopCardinality or as the cardinality of a specific collection-valued data item of the data input of the MI Activity. The latter is described in detail below.

The number of *instances* to be generated is evaluated once. Subsequently the number of *instances* are generated. If the *instances* are generated sequentially, a new *instance* is generated only after the previous has been completed. Otherwise, multiple *instances* to be executed in parallel are generated.

Attributes are available to support the different possibilities of behavior. The completionCondition Expression is a Boolean predicate that is evaluated every time an *instance* completes. When evaluated to *true*, the remaining *instances* are cancelled, a *token* is produced for the *outgoing* Sequence Flow, and the MI Activity completes.

The attribute behavior defines if and when an Event is thrown from an Activity *instance* that is about to complete. It has values of none, one, all, and complex, assuming the following behavior:

- **none**: an EventDefinition is thrown for all *instances* completing.
- **one**: an EventDefinition is thrown upon the first *instance* completing.
- **all**: no Event is ever thrown.

- **complex**: the complexBehaviorDefinitions are consulted to determine if and which Events to throw.

For the behaviors of none and one, an EventDefinition (which is referenced from MultipleInstanceLoopCharacteristics through the noneEvent and oneEvent associations, respectively) is thrown which automatically carries the current runtime attributes of the MI Activity. That is, the ItemDefinition of these SignalEventDefinitions is implicitly given by the specific runtime attributes of the MI Activity.

The complexBehaviorDefinition association references multiple ComplexBehaviorDefinition entities which each point to a Boolean condition being a FormalExpression and an Event which is an ImplicitThrowEvent. Whenever an Activity *instance* completes, the conditions of all ComplexBehaviorDefinitions are evaluated. For each ComplexBehaviorDefinition whose condition is evaluated to *true*, the associated Event is automatically thrown. That is, a single Activity completion may lead to multiple different Events that are thrown. The Events may then be caught on the boundary of the MI Activity. Multiple ComplexBehaviorDefinitions offer an easy way of implicitly spawning different flow at the MI Activity boundary for different situations indicating different states of progress in the course of executing the MI Activity.

The completionCondition, the condition in the ComplexBehaviorDefinition, and the DataInputAssociation of the Event in the ComplexBehaviorDefinition can refer to the MI Activity *instance* attributes and the loopDataInput, loopDataOutput, inputDataItem, and outputDataItem that are referenced from the MultiInstanceLoopCharacteristics.

In practice, a MI Activity is executed over a data *collection*, processing as input the data values in the *collection* and producing as *output* data values in a *collection*. The *input* data collection is passed to the MI outer Activity's loopDataInput from a Data Object in the Process *scope* of the MI Activity. Under BPMN data flow constraints, the Data Object is linked to MI activity's loopDataInput through a DataInputAssociation. To indicate that the Data Object is a *collection*, its respective symbol is marked with the MI indicator (three-bar). The items of the loopDataInput *collection* are used to determine the number of *instances* required to be executed (whether sequentially or in parallel). Accordingly, the inner *instances* are created and data values from the loopDataInput are extracted and assigned to the respective *instances*. Specifically, the values from the loopDataInput items are passed to an inputDataItem, created in the scope of the outer Activity. The value in the inputDataItem can be passed to the loopDataInput of each inner *instance*, where a DataInputAssociation links both. The process of extraction is left under-specified. In practice, it would entail a special-purpose mediator which not only provides the extraction and data assignment, but also any required data transformation.

Each *instance* processes the data value of its DataInput. It produces a value in its DataOutput if it completes successfully. The DataOutPut value of the *instance* is passed to a corresponding outputDataItem in the outer Activity, where a DataOutputAssociation links both. Each outputDataItem value is updated in the loopDataOutput *collection*, in the corresponding item. The mechanism of this update is left underspecified, and again would be implemented through a special purpose mediator. The loopDataOutput is passed to the MI Activity's Process *scope* through a Data Object that has a DataOutputAssociation linking both.

It should be noted that the *collection* in the Process *scope* should not be accessible until all its items have been written to. This is because, it could be accessed by an Activity running concurrently, and therefore control flow through *token* passing cannot guarantee that the *collection* is fully written before it is accessed.

The MI Activity is *compensated* only if all its *instances* have completed successfully.

Workflow Patterns Support: WCP-21 Structured Loop, Multiple Instance Patterns WCP 13, 14, 34, 36

14.3. Gateways

This section describes the behavior of Gateways.

14.3.1. Parallel Gateway (Fork and Join)

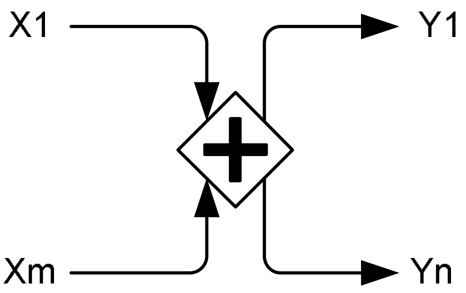


Figure 14-3 – Merging and Branching Sequence Flow for a Parallel Gateway

On the one hand, the Parallel Gateway is used to synchronize multiple concurrent branches (merging behavior). On the other hand, it is used to spawn new concurrent threads on parallel branches (branching behavior).

Table 14-1 – Parallel Gateway Execution Semantics

Operational Semantics	The parallel gateway is activated if there is at least one Token on each incoming sequence flow. The parallel gateway consumes exactly one Token from each incoming sequence flow and produces exactly one Token at each outgoing sequence flow. If there are excess Tokens at an incoming sequence flow, these Tokens remain at this sequence flow after execution of the gateway.
Exception Issues	The parallel gateway cannot throw any exception.
Workflow Patterns Support	Parallel Split (WCP-2) Synchronization (WCP-3)

14.3.2. Exclusive Gateway (Exclusive Decision (data-based) and Exclusive Merge)

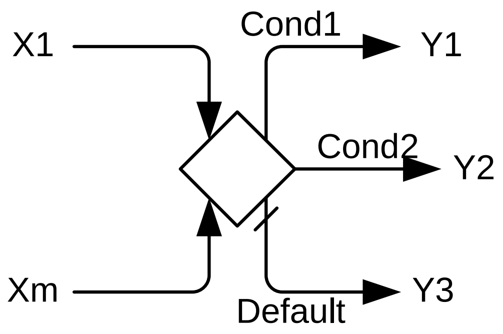


Figure 14-4 – Merging and Branching Sequence Flow for an Exclusive Gateway

The Exclusive Gateway has pass-through semantics for a set of incoming branches (merging behavior). Further on, each activation leads to the activation of exactly one out of the set of outgoing branches (branching behavior).

Table 14-2 – Exclusive Gateway Execution Semantics

Operational Semantics	Each Token arriving at any incoming sequence flow activates the gateway and is routed to exactly one of the outgoing sequence flow. In order to determine the outgoing sequence flow that receives the Token, the conditions are evaluated in order. The first condition that evaluates to true determines the sequence flow the Token is sent to. No more conditions are henceforth evaluated. If and only if none of the conditions evaluates to true, the Token is passed on the default sequence flow. In case all conditions evaluate to false and a default flow has not been specified, an exception is thrown.
Exception Issues	The exclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.
Workflow Patterns Support	Exclusive Choice (WCP-4) Simple Merge (WCP-5) Multi-Merge (WCP-8)

14.3.3. Inclusive Gateway (Inclusive Decision and Inclusive Merge)

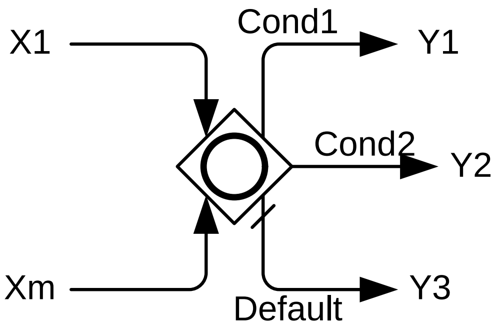


Figure 14-5 – Merging and Branching Sequence Flow for an Inclusive Gateway

The Inclusive Gateway synchronizes a certain subset of branches out of the set of concurrent incoming branches (merging behavior). Further on, each firing leads to the creation of threads on a certain subset out of the set of outgoing branches (branching behavior).

Table 14-3 – Inclusive Gateway Execution Semantics

Operational Semantics	The Inclusive Gateway is activated if • At least one incoming sequence flow has at least one *Token* and • for each empty incoming sequence flow, there is no *Token* in the graph anywhere upstream of this sequence flow, i.e., there is no directed path (formed by Sequence Flow) from a *Token* to this sequence flow unless the path visits the inclusive gateway or the path visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway. Upon execution, a *Token* is consumed from each incoming sequence flow that has a *Token*. A *Token* will be produced on some of the outgoing sequence flows. In order to determine the outgoing sequence flows that receive a *Token*, all conditions are evaluated. The evaluation does not have to respect a certain order. For every condition, which evaluates to true, a *Token* must be passed on the respective sequence flow. If and only if none of the conditions evaluates to true, the *Token* is passed on the default sequence flow. In case all conditions evaluate to false and a default flow has not been specified, the inclusive gateway throws an exception.

Exception Issues	The inclusive gateway throws an exception in case all conditions evaluate to false and a default flow has not been specified.
Workflow Patterns Support	Multi-Choice (WCP-6) Structured Synchronizing Merge (WCP-7) Acyclic Synchronizing Merge (WCP-37) General Synchronizing Merge (WCP-38)

14.3.4. Event-based Gateway (Exclusive Decision (event-based))

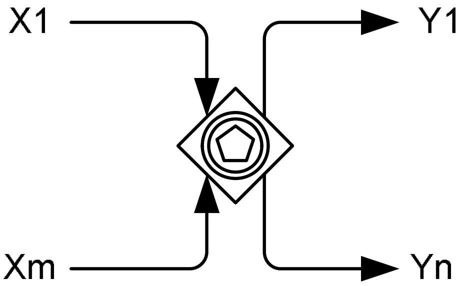


Figure 14-6 – Merging and branching Sequence Flow for an Event-Based Gateway

The Event-Based Gateway has pass-through semantics for a set of incoming branches (merging behavior). Exactly one of the outgoing branches is activated afterwards (branching behavior), depending on which of Events of the Gateway configuration is first triggered. The choice of the branch to be taken is deferred until one of the subsequent Tasks or Events completes. The first to complete causes all other branches to be withdrawn.

When used at the Process start as a Parallel Event Gateway, only message-based triggers are allowed. The Message *triggers* that are part of the Gateway configuration must be part of a Conversation with the same correlation information. After the first *trigger* instantiates the Process, the remaining Message *triggers* will be a part of the Process *instance* that is already active (rather than creating new Process *instances*).

Table 14-4 – Event-Based Gateway Execution Semantics

Exception Issues	The event-based gateway cannot throw any exception.
Workflow Patterns Support	Deferred Choice (WCP-16)

14.3.5. Complex Gateway (related to Complex Condition and Complex Merge)

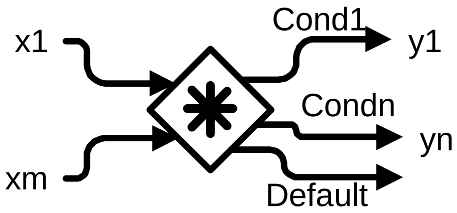


Figure 14-7 – Merging and branching Sequence Flow for a Complex Gateway

The Complex Gateway facilitates the specification of complex synchronization behavior, in particular race situations. The diverging behavior is similar to the Inclusive Gateway. Each incoming gate of the Complex Gateway has an attribute activationCount, which can be used in an expression as an integer-valued variable. This variable represents the number of *tokens* that are currently on the respective *incoming* Sequence Flow. The Complex Gateway has an attribute activationExpression. An activationExpression is a Boolean Expression that refers to data and to the activationCount of incoming gates. For example, an activationExpression could be x1+x2+…+xm >= 3 stating that it needs 3 out of the m incoming gates to have a *token* in order to proceed. To prevent undesirable oscillation of activation of the Complex Gateway, ActivationCount variables should only be used in subexpressions of the form *expr* >= *const* where *expr* is an arithmetic Expression that uses only addition and *const* is an Expression whose evaluation remains constant during execution of the Process.

Each *outgoing* Sequence Flow of the Complex Gateway has a Boolean condition that is evaluated to determine whether that Sequence Flow receives a *token* during the execution of the Gateway. Such a condition may refer to internal state of the Complex Gateway. There are two states: waiting for start (represented by the runtime attribute waitingForStart = *true*) and waiting for reset (waitingForStart=*false*).

Table 14-5 – Semantics of the Complex Gateway

Operational Semantics	The Complex Gateway is in one of the two states: *waiting for start* or *waiting for reset*, initially it is in *waiting for start*. If it is *waiting for start*, then it waits for the `activationExpression` to become *true*. The `activationExpression` is not evaluated before there is at least one *token* on some *incoming* Sequence Flow. When it becomes *true*, a *token* is consumed from each *incoming* Sequence Flow that has a *token*. To determine which *outgoing* Sequence Flow receive a *token*, all conditions on the *outgoing* Sequence Flow are evaluated (in any order). Those and only those which evaluate to *true* receive a *token*. If no condition evaluates to *true*, and only then, the *default* Sequence Flow receives a *token*. If no default flow is specified an exception is thrown. The Gateway changes its state to *waiting for reset*. The Gateway remembers from which of the *incoming* Sequence Flow it consumed *tokens* in the first phase. When *waiting for reset*, the Gateway waits for a *token* on each of those *incoming* Sequence Flow from which it has not yet received a *token* in the first phase unless such a *token* does not come (cf. inclusive join behavior). More precisely, the Gateway being *waiting for reset*, *resets* when for each *incoming* Sequence Flow from which no *token* was consumed in the first phase, there is either a *token* on that Sequence Flow or there is no *token* in the graph anywhere upstream of this Sequence Flow, i.e., there is no directed path (formed by Sequence Flow) from a *token* to this Sequence Flow unless • the path visits the Complex Gateway or • the path visits a node that has a directed path to a non-empty *incoming* Sequence Flow of the Complex Gateway or to an *incoming* Sequence Flow from which a *token* was consumed in the first phase. When the Gateway resets, it consumes a *token* from each *incoming* Sequence Flow that has a *token* and from which it had not yet consumed a *token* in the first phase. It then evaluates all conditions on the *outgoing* Sequence Flow (in any order) to determine which Sequence Flow receives a *token*. Those and only those which evaluate to *true* receive a *token*. If no condition evaluates to *true*, and only then, the *default* Sequence Flow receives a *token*. The Gateway changes its state back to the state *waiting for start*. Note that the Gateway might not produce any *tokens* in this phase and no exception is thrown. Note that the conditions on the *outgoing* Sequence Flow may evaluate differently in the two phases, e.g., by referring to the state of the Gateway (runtime attribute `waitingForStart`).

Exception issues	The Complex Gateway throws an exception when it is activated in the state *waiting for start*, no condition on any *outgoing* Sequence Flow evaluates to *true* and no *default* Sequence Flow is specified.
Workflow Patterns Support	Structured Discriminator (WCP-9) Blocking Discriminator (WCP-28) Structured Partial Join (WCP-30) Blocking Partial Join (WCP-31)

14.4. Events

This section describes the handling of Events.

14.4.1. Start Events

For single Start Events, handling consists of starting a new Process *instance* each time the Event occurs. Sequence Flow leaving the Event is then followed as usual.

If the Start Event participates in a Conversation that includes other Start Events, a new Process *instance* is only created if none already exists for the specific Conversation (identified through its associated correlation information) of the Event occurrence.

A Process can also be started via an Event-Based Gateway. In that case, the first matching Event will create a new *instance* of the Process, and waiting for the other Events originating from the same decision stops, following the usual semantics of the Event-Based Exclusive Gateway. Note that this is the only scenario where a Gateway can exist without an *incoming* Sequence Flow.

It is possible to have multiple groups of Event-Based Gateways starting a Process, provided they participate in the same Conversation and hence share the same correlation information. In that case, one Event out of each group needs to arrive; the first one creates a new Process *instance*, while the subsequent ones are routed to the existing *instance*, which is identified through its correlation information.

14.4.2. Intermediate Events

For Intermediate Events, the handling consists of waiting for the Event to occur. Waiting starts when the Intermediate Event is reached. Once the Event occurs, it is consumed. Sequence flow leaving the Event is followed as usual.

14.4.3. Intermediate Boundary Events

For boundary Events, handling first consists of consuming the Event occurrence. If the cancelActivity attribute is set, the Activity the Event is attached to is then cancelled (in case of a multi-instance, all its *instances* are cancelled); if the attribute is not set, the Activity continues execution (only possible for Message, Signal, Timer and Conditional Events, not for Error Events). Execution then follows the Sequence Flow connected to the boundary Event.

14.4.4. Event Sub-Processes

Event Sub-Processes allow to handle an Event within the context of a given Sub-Processes or Process. An Event Sub-Process always begins with a Start Event, followed by Sequence Flow. Event Sub-Processes are a special kind of Sub-Process: they create a scope and are instantiated like a Sub-Process, but they are not instantiated by normal control flow but only when the associated Start Event is triggered. Event Sub-Processes are self-contained and must not be connected to the rest of the Sequence Flow in the Sub-Processes; also they cannot have attached boundary Events. They run in the context of the Sub-Process, and thus have access to its context.

An Event Sub-Process cancels execution of the enclosing Sub-Process, if the isInterrupting attribute of its Start Event is set; for a multi-instance Activity this cancels only the affected *instance*. If the isInterrupting attribute is not set (not possible for an Error Event Sub-Processes), execution of the enclosing Sub-Process continues in parallel to the Event Sub-Process.

An Event Sub-Process can optionally retrigger the Event through which it was triggered, to cause its continuation outside the boundary of the associated Sub-Process. In that case the Event Sub-Process is performed when the Event occurs; then control passes to the boundary Event, possibly canceling the Sub-Process (including running handlers).

Operational semantics

- A non-interrupting Event Sub-Process becomes initiated, and thus *Enabled* and *Running*, through the Activity to which it is attached. The non-interrupting *Event Handler* may only be initiated after the parent Activity is *Running*. More than one non-interrupting *Event Handler* may be initiated and they may be initiated at different times. There might be multiple *instances* of the non-interrupting *Event Handler* at a time.

- An Event Sub-Process completes when all *tokens* have reached an End Event, like any other Sub-Process. If the *parent* Activity enters the state *Completing*, it remains in that state until all contained active Event Sub-Processes have completed. While the *parent* Activity is in that *Completing*, no new Event Sub-Processes can be initiated.

14.4.5. Compensation

Compensation is concerned with undoing steps that were already successfully completed, because their results and possibly side effects are no longer desired and need to be reversed. If an Activity is still active, it cannot be compensated, but rather needs to be canceled. Cancellation in turn may result in *compensation* of already successfully completed portions of an active Activity, in case of a Sub-Process.

Compensation is performed by a *compensation handler.* A *compensation handler* can either be a Compensation Event Sub-Process (for a Sub-Process or Process), or an associated Compensation Activity (for any Activity). A *compensation handler* performs the steps necessary to reverse the effects of an Activity. In case of a Sub-Process, its Compensation Event Sub-Process has access to Sub-Process data at the time of its completion ("snapshot data").

Compensation is triggered by a *throw* Compensation Event, which typically will be raised by an *error handler*, as part of cancellation, or recursively by another *compensation handler*. That Event specifies the Activity for which *compensation* is to be performed, either explicitly or implicitly.

Compensation Handler

A *compensation handler* is a set of Activities that are not connected to other portions of the BPMN model. The *compensation handler* starts with a *catch* Compensation Event. That *catch* Compensation Event either is a boundary Event, or, in case of a Compensation Event Sub-Process, the handler's Start Event.

A *compensation handler* connected via a boundary Event can only perform "black-box" *compensation* of the original Activity. This *compensation* is modeled with a specialized Compensation Activity.

A Compensation Event Sub-Process is contained within a Process or a Sub-Processes. It can access data that is part of its parent, snapshot at the point in time when its parent has completed. A *compensation* Event Sub-Process can in particular recursively trigger *compensation* for Activities contained in that its parent.

It is possible to specify that a Sub-Process can be compensated without having to define the *compensation handler*. The Sub-Process attribute compensable, when set, specifies that default *compensation* is implicitly defined, which recursively compensates all successfully completed Activities within that Sub-Process, invoking them in reverse order of their forward execution.

Compensation Triggering

Compensation is triggered using a *throw* Compensation Event, which can either be an Intermediate or an End Event. The Activity which needs to be compensated is referenced. If the Activity is clear from the context, it doesn't have to be specified and defaults to the current Activity. A typical scenario for that is an inline *error handler* of a Sub-Process that cannot recover the *error*, and as a result would trigger *compensation* for that Sub-Process. If no Activity is specified in a "global" context, all completed Activities in the Process are compensated.

By default, *compensation* is triggered synchronously, that is, the *throw* Compensation Event waits for the completion of the triggered *compensation handler*. Alternatively, *compensation* can just be triggered without waiting for its completion, by setting the *throw* Compensation Event's waitForCompletion attribute to *false*.

Multiple *instances* typically exist for Loop or Multi-Instance Sub-Processes. Each of these has its own *instance* of its Compensation Event Sub-Process, which has access to the specific snapshot data that was current at the time of completion of that particular *instance*. Triggering *compensation* for the Multi-Instance Sub-Process individually triggers *compensation* for all *instances* within the current *scope*. If *compensation* is specified via a boundary *compensation handler*, this boundary *compensation handler* also is invoked once for each *instance* of the Multi-Instance Sub-Process in the current *scope*.

Relationship between Error Handling and Compensation

Compensation employs a "presumed abort principle", which has a number of consequences. First, only completed Activities are compensated; *compensation* of a failed Activity results in an empty operation. Thus, when an Activity fails, i.e., is left because an *error* has been thrown, it's the *error handler's* responsibility to ensure that no further *compensation* will be necessary once the *error handler* has completed. Second, if no *error* Event Sub-Process is specified for a particular Sub-Process and a particular *error*, the default behavior is to automatically call *compensation* for all contained Activities of that Sub-Process if that *error* occurs, thus ensuring the "presumed abort" invariant.

Operational Semantics

- A Compensation Event Sub-Process becomes enabled when its *parent* Activity transitions into state *Completed*. At that time, a snapshot of the data associated with the parent Activity is taken and kept for later usage by the Compensation Event Sub-Process. In case the *parent* Activity is a *multi-instance* or *loop*, for each *instance* a separate data snapshot is taken, which is used when its associated Compensation Event Sub-Process is triggered.
- When *compensation* is triggered for the *parent* Activity, its Compensation Event Sub-Process is activated and runs. The original context data of the *parent* Activity is restored from the data snapshot. In case the *parent* Activity is a multi-instance or loop, for each *instance* the dedicated snapshot is restored and a dedicated Compensation Event Sub-Process is activated.
- An associated Compensation Activity becomes enabled when the Activity it is associated with transitions into state *Completed*. When *compensation* is triggered for that Activity, the associated Compensation Activity is activated. In case the Activity is a multi-instance or loop, the Compensation Activity is triggered only once, too, and thus has to compensate the effects of all *instances*.
- Default compensation ensures that Compensation Activities are performed in reverse order of the execution of the original Activities, allowing for concurrency when there was no dependency between the original Activities. Dependencies between original Activities that default compensation must consider are the following
  - A Sequence Flow between Activities A and B results in compensation of B to be performed before compensation of A.
  - A data dependency between Activities A and B, e.g., through an IORules specification in B referring to data produced by A, results in compensation of B to be performed before compensation of A.
  - If A and B are two Activities that were active as part of an Ad-Hoc Sub-Process, then compensation of B must be performed before compensation of A if A completed before B started.
  - *Instances* of a loop or sequential multi-instance are compensated in reverse order of their forward completion. *Instances* of a parallel multi-instance can be compensated in parallel.
  - If a Sub-Process A has a *boundary* Event connected to Activity B, then compensation of B must be performed before compensation of A if that particular Event occurred. This also applies to multi-instances and loops.

14.4.6. End Events

Process level end events

For a “terminate” End Event, the Process is abnormally terminated.

For all other End Events, the behavior associated with the Event type is performed, e.g., the associated Message is sent for a Message End Event, the associated signal is sent for a Signal End Event, and so on. The Process *instance* is then completed, if and only if the following two conditions hold:

- All start nodes of the Process have been visited. More precisely, all Start Events have been triggered, and for all starting Event-Based Gateways, one of the associated Events has been triggered.

- There is no *token* remaining within the Process *instance*.

Sub-process level end events

For a “terminate” End Event, the Sub-Process is abnormally terminated. In case of a multi-instance Activity, only the affected *instance* is terminated.

For a “cancel” End Event, the Sub-Process is abnormally terminated and the associated transaction is aborted. Control leaves the Sub-Process through a cancel intermediate boundary Event.

For all other End Events, the behavior associated with the Event type is performed, e.g., the associated Message is sent for a Message End Event, the associated signal is sent for a signal End Event, and so on. The Sub-Process *instance* is then completed, if and only if the following two conditions hold:

- All start nodes of the Sub-Process have been visited. More precisely, all Start Events have been triggered, and for all starting Event-Based Gateways, one of the associated Events has been triggered.
- There is no *token* remaining within the Sub-Process *instance*.

15. Mapping BPMN Models to WS-BPEL

Note: The contents of this chapter is required for BPMN **BPEL Process Execution Conformance** or for BPMN **Complete Conformance** . However, this chapter is not required for BPMN **Process Modeling Conformance**, BPMN **Process Choreography Conformance**, or BPMN **Process Execution Conformance**. For more information about BPMN conformance types, see page 28.

This chapter covers a mapping of a BPMN model to WS-BPEL that is derived by analyzing the BPMN objects and the relationships between these objects.

A Business Process Diagram can be made up of a set of (semi-) independent components, which are shown as separate Pools, each of which represents an orchestration Process. There is not a specific mapping of the diagram itself, but rather, each of these *orchestration* Processes maps to an individual WS-BPEL *process*.

Not all BPMN *orchestration* Processes can be mapped to WS-BPEL in a straight-forward way. That is because BPMN allows the modeler to draw almost arbitrary graphs to model control flow, whereas in WS-BPEL, there are certain restrictions such as control-flow being either block-structured or not containing cycles. For example, an unstructured *loop* cannot directly be represented in WS-BPEL.

To map a BPMN orchestration Process to WS-BPEL it must be *sound*, that is it must contain neither a *deadlock* nor a *lack of synchronization.* A deadlock is a reachable state of the Process that contains a *token* on some Sequence Flow that cannot be removed in any possible future. A lack of synchronization is a reachable state of the Process where there is more than one *token* on some Sequence Flow. For further explanation of these terms, we refer to the literature. To define the structure of BPMN Processes, we introduce the following concepts and terminology. The Gateways and the Sequence Flow of the BPMN orchestration Process form a directed graph. A *block* of the diagram is a connected sub-graph that is connected to the rest of the graph only through exactly two Sequence Flow: exactly one Sequence Flow entering the block and exactly one Sequence Flow leaving the block. A *block hierarchy* for a Process model is a set of blocks of the Process model in which each pair of blocks is either nested or disjoint and which contains the maximal block (i.e. the whole Process model) A block that is nested in another block B is also called a *subblock* of *B* (cf. Figure 15-1). Each block of the block hierarchy of a given BPMN orchestration Process has a certain structure (or pattern) which provides the basis for defining the BPEL mapping.

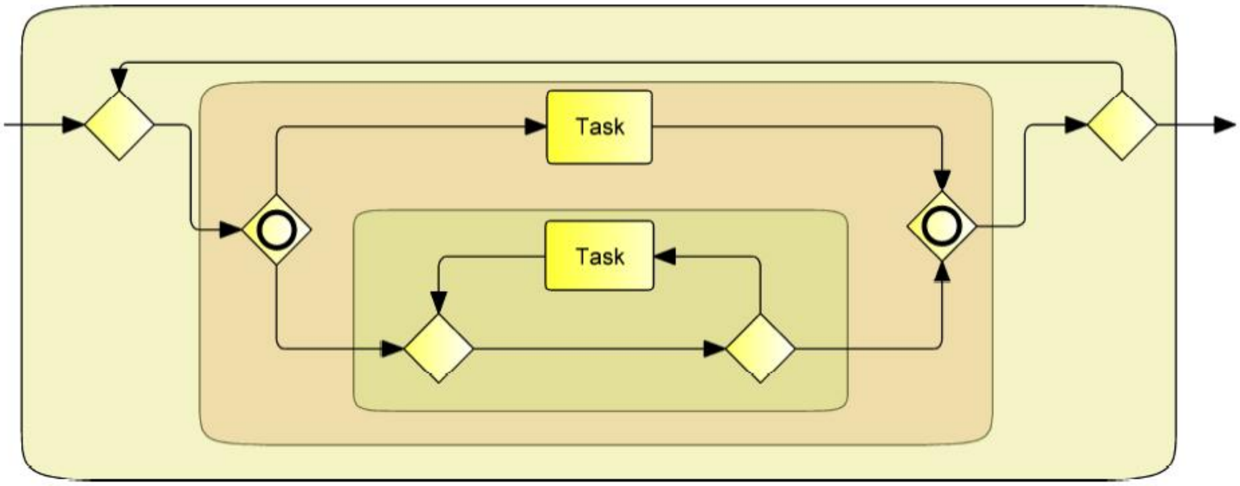


Figure 15-1 – A BPMN orchestration process and its block hierarchy

The following sections define a syntactical BPEL mapping prescribing the resulting BPEL model at the syntactical level, and a semantic BPEL mapping prescribing the resulting BPEL model in terms of its observable behavior. The syntactical BPEL mapping is defined for a subset of BPMN models based on certain patterns of BPMN blocks, whereas the semantical BPEL mapping (which extends the syntactical mapping) does not enforce block patterns, allowing for the mapping a larger class of BPMN models without prescribing the exact syntactical representation in BPEL.

15.1. Basic BPMN-BPEL Mapping

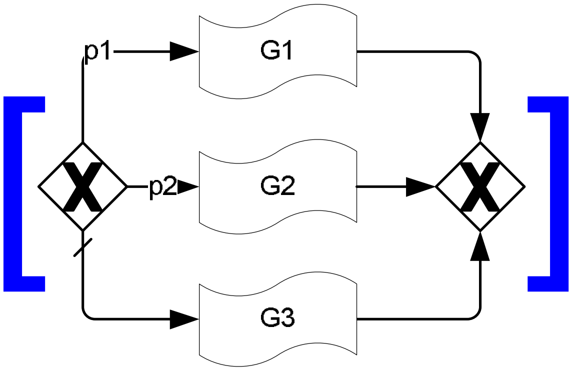
This section introduces a partial mapping function from BPMN orchestration Process models to WS-BPEL executable Process models by recursively defining the mapping for elementary BPMN constructs such as Tasks and Events, and for blocks following the patterns described here. Mapping a BPMN block to WS-BPEL includes mapping all of its associated attributes. The observable behavior of a WS-BPEL process resulting from a BPEL mapping is the same as that of the original BPMN orchestration Process.

We use the notation [BPMN construct] to denote the WS-BPEL construct resulting from mapping the BPMN construct.

Examples are

[ServiceTask] = Invoke Activity

which says that a BPMN Service Task is mapped to a WS-BPEL Invoke Activity, or



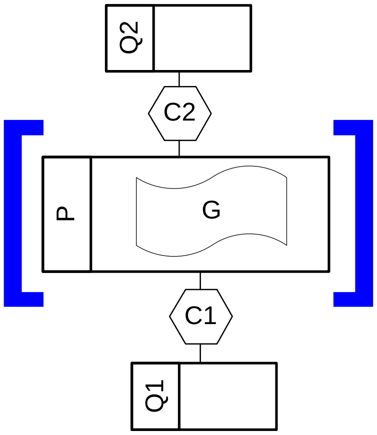
=

```
<if><condition>[p1]</condition>
  [G1]
<elseif><condition>[p2]</condition>
  [G2]
</elseif>
<else>
  [G3]
</else>
</if>
```

which says that the data-based exclusive choice controlled by the two predicates p1 and p2, containing the three BPMN blocks G1, G2 and G3 is mapped to the WS-BPEL on the right hand side, which recursively uses the mappings of those predicates and those sub-graphs. Note that we use the "waved rectangle" symbol throughout this section to denote BPMN blocks.

15.1.1. Process

The following figure describes the mapping of a Process, represented by its defining Collaboration, to WS-BPEL. The process itself is described by a contained graph G of flow elements) to WS-BPEL. The Process interacts with *Participants* Q1…Qn via Conversations C1…Cm:



=

```
<process name="[P-name]"
    targetNamespace="[targetNamespace]"
    expressionLanguage="[expressionLanguage]"
    suppressJoinFailure="yes"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
  <partnerLinks>
    [ {P-Interfaces} UNION {Qi-Interfaces} ]
  </partnerLinks>
  <variables>
    [ {dataObjects} UNION {properties} ]
  </variables>
  <correlationSets>
    [ {Ci-CorrelationKeys} ]
  </correlationSets>
  [G]
</process>
```

The partner links of the corresponding WS-BPEL process are derived from the set of interfaces associated with each participant. Each interface of the *Participant* containing the Process P itself is mapped to a WS-BPEL partner link with a "myRole" specification, each interface of each other *Participant* Qi is mapped to a WS-BPEL partner link with a "partnerRole" specification.

The variables of the corresponding WS-BPEL process are derived from the set "{dataObjects}" of all Data Objects occurring within G, united with the set "{properties}" of all properties occurring within G, without Data Objects or properties contained in nested Sub-Processes. See Section "Handling Data" on page 467 for more details of this mapping.

The correlation sets of the corresponding WS-BPEL process are derived from the CorrelationKeys of the set of Conversations C1…Cn.. See page 451 for more details of this mapping.

15.1.2. Activities

Common Activity Mappings

The following table displays a set of mappings of general BPMN Activity attributes to WS-BPEL activity attributes.

Table 15-1 – Common Activity Mappings to WS-BPEL

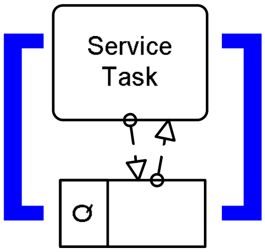
Activity	Mapping to WS-BPEL
name	The name attribute of a BPMN activity is mapped to the name attribute of a WS-BPEL activity by removing all characters not allowed in an XML NCName, and ensuring uniqueness by adding an appropriate suffix. In the subsequent diagrams, this mapping is represented as [name].

Task Mappings

The following sections contain the mappings of the variations of a Task to WS-BPEL.

Service Task

The following figure shows the mapping of a Service Task to WS-BPEL:

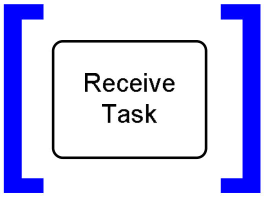


```
  <invoke name="[Task-name]"
    partnerLink="[Q, Task-operation-interface]"
=   portType="[Task-operation-interface]"
    operation="[Task-operation]">
  </invoke>
```

The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the Service Task is connected to by Message Flow, and from the interface referenced by the operation of the Service Task.

Receive Task

The following figure shows the mapping of a Receive Task to WS-BPEL:

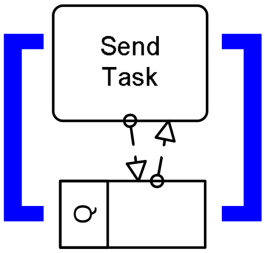


```
  <receive name="[Task-name]"
    createInstance="[instantiate? 'yes':'no']"
=   partnerLink="[Task-operation-interface]"
    portType="[Task-operation-interface]"
    operation="[Task-operation]">
  </receive>
```

The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the Receive Task.

Send Task

The following figure shows the mapping of a Send Task to WS-BPEL:

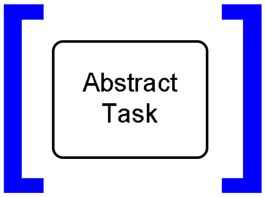


```
  <invoke name="[Task-name]"
    partnerLink="[Q, Task-operation-interface]"
=   portType="[Task-operation-interface]"
    operation="[Task-operation]">
  </invoke>
```

The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the Send Task is connected to by Message Flow, and from the interface referenced by the operation of the Send Task.

Abstract Task

The following figure shows the mapping of an Abstract Task to WS-BPEL:



```
= <empty name="[Task-name]">
  </empty>
```

Service Package

Message

For Messages with a scalar data item definition typed by an XML schema definition, the following figure shows the mapping to WS-BPEL, using WSDL 1.1:

```
[ <Message name="msg-name">
    <StructureDefinition typeLanguage=
         "http://www.w3.org/2001/XMLSchema">
      xmlSchema
    </StructureDefinition>
  </Message> ]
```

```
= <wsdl:message name="[msg-name]">
    [xmlSchema]
  </wsdl:message>
```

The top-level child elements of the XML schema defining the structure of the BPMN Message are mapped to the WSDL's message's parts.

Interface and Operation

The following figure shows the mapping of a BPMN interface with its operations to WS-BPEL, using WSDL 1.1:

```
<Interface name="if-name">
  <Operations>
    <Operation name="op1-name">
      <inMessageRef ref="msg1i-name"/>
      <outMessageRef ref="msg1o-name"/>
      <errorRef ref="error1a-name"/>
      ...
    </Operation>
    ...
  </Operations>
</Interface>
```

=

```
<wsdl:portType name="[if-name]">
  <operation name="[op1-name]">
    <wsdl:input message="[msg1i-name]" />
    <wsdl:output message="[msg1o-name]" />
    <wsdl:fault name="[error1a-faultname]"
        message="[error1a-name]" />
    ...
  </operation>
  ...
</wsdl:portType>
```

Conversations and Correlation

For those BPMN nodes sending or receiving Messages (i.e., Message Events, Service, send or Receive Tasks) that have an associated key-based Correlation Key, the mapping of that key-based Correlation Key is as follows:

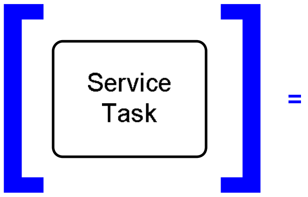
```
<KeyBasedCorrelationSet name="c-set">
  <Key name="k-name1" type="k-type1"
        messageRef="msg-name1">
    <MessageKeyExpression
        expressionLanguage="lang1">
      expr1
    </MessageKeyExpression>
  </Key>
  ...
  <Key name="k-nameN" />
    ...
</KeyBasedCorrelationSet>
```

=

```
<vprop:property name="[k-name1]" type="[k-type1]"/>
...
<vprop:property name="[k-nameN]" />

<vprop:propertyAlias propertyName="[kName1]"
    messageType="[msg-name1]"
    part="[expr1-part]">
  <vprop:query queryLanguage="[lang1]">
    [expr1]
  </vprop:query>
</vprop:propertyAlias>
...
<vprop:propertyAlias propertyName="[kNameN]" />

<correlationSets>
  <correlationSet name="[c-set]"
      properties="[k-name1] ... [k-nameN]"/>
  ...
</correlationSets>
```

The messageType of the BPEL property alias is appropriately derived from the itemDefinition of the Message referenced by the BPMN Message key expression. The name of the Message part is derived from the Message key expression. The Message key expression itself is transformed into an expression relative to that part.

The mapping of Activities with an associated key-based Correlation Key is extended to reference the above BPEL correlation set in the corresponding BPEL correlations element. The following figure shows that mapping in the case of a Service Task with an associated key-based Correlation Key:



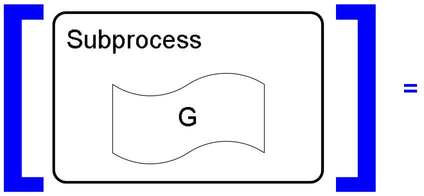
```
<invoke name="[Task-name]"
  partnerLink="[Q, Task-operation-interface]"
  portType="[Task-operation-interface]"
  operation="[Task-operation]">
  <correlations>
    <correlation set="[Task-messageFlow-conversation-correlationKey]"
        initiate="[initialInConversation? 'join':'no']"/>
  </correlations>
</invoke>
```

The initiate attribute of the BPEL `correlation` element is set depending on whether or not the associated Message Flow initiates the associated Conversations, or participates in an already existing Conversation. If there are multiple `CorrelationKeys` associated with the Conversation, multiple *correlation* elements are used.

Sub-Process Mappings

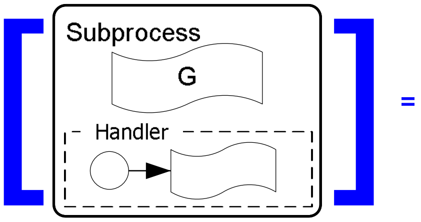
The following table displays the mapping of an embedded Sub-Process with Adhoc=”*False*” to a WS-BPEL scope. (This extends the mappings that are defined for all Activities--see page 450):

The following figure shows the mapping of a BPMN Sub-Process without an Event Sub-Process:



```
<scope>
  [G]
</scope>
```

The following figure shows the mapping of a BPMN Sub-Process with an Event Sub-Process. (Event Sub-Processes could also be added to a top-level Process, in which case their mapping extends correspondingly.)



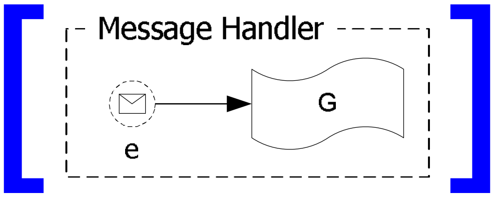
```
<scope>
  [Handler]
  [G]
</scope>
```

Note that in case of multiple Event Sub-Processes, there would be multiple WS-BPEL handlers.

Mapping of Event Sub-Processes

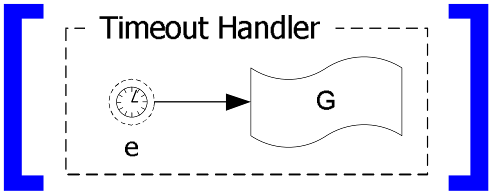
Note that if a Sub-Process contains multiple Event Sub-Processes, all become handlers of the associated WS-BPEL scope, ordered and grouped as specified by WS-BPEL.

Non-interrupting Message Event Sub-Processes are mapped to WS-BPEL event handlers as follows:



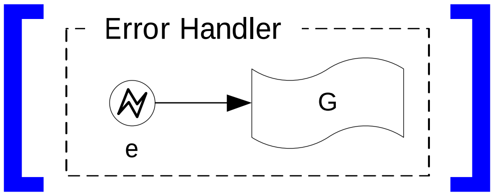
```
=   <eventHandlers>
      <onEvent partnerLink="[e-operation-interface]"
            operation="[e-operation]">
        <scope>[G]</scope>
      </onEvent>
    </eventHandlers>
```

Timer Event Sub-Processes are mapped to WS-BPEL event handlers as follows:



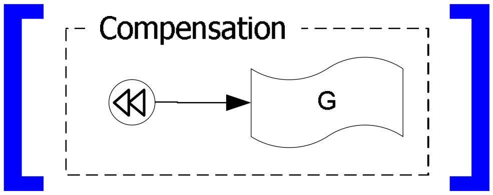
```
=   <eventHandlers>
      <onAlarm>[timer-spec]
        <scope>[G]</scope>
      </onAlarm>
    </eventHandlers>
```

Error Event Sub-Processes are mapped to WS-BPEL fault handlers as follows:



```
=   <faultHandlers>
      <catch faultName="[e-fault]">
        [G]
      </catch>
    </faultHandlers>
```

A Compensation Event Sub-Process is mapped to a WS-BPEL compensation handler as follows:



```
=   <compensationHandler>
      [G]
    </compensationHandler>
```

Activity Loop Mapping

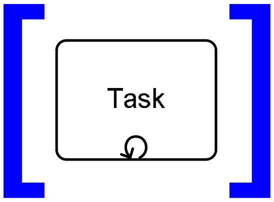
Standard *loops* with a testTime attribute “Before” or “After” execution of the Activity map to WS-BPEL `while` and `repeatUntil` activities in a straight-forward manner. When the LoopMaximum attribute is used, additional activities are used to maintain a *loop* counter.

Multi-instance Activities map to WS-BPEL `forEach` activities in a straight-forward manner.

Standard Loops

The mappings for standard *loops* to WS-BPEL are described in the following.

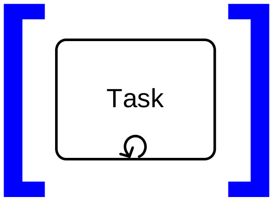
A standard *loop* with testTime= “Before” maps to WS-BPEL as follows, where *p* denotes the *loop* condition:



=

```
<while>
  <condition>[p]</condition>
  [Task]
</while>
```

A standard *loop* with testTime= “After” maps as follows, where *p* denotes the *loop* condition:



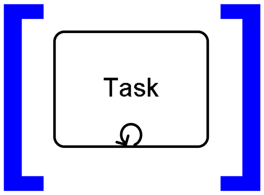
=

```
<repeatUntil>
  [Task]
  <condition>[not p]</condition>
</repeatUntil>
```

Dealing with LoopMaximum

When the `LoopMaximum` attribute is specified for an `Activity`, the *loop* requires additional set up for maintaining a counter.

A standard *loop* with testTime=“Before” and a `LoopMaximum` attribute maps to WS-BPEL as follows (again, *p* denotes the `loopCondition`):

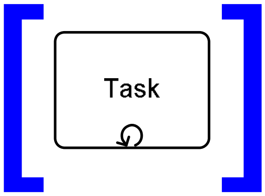


=

```
<variable name="[counter]" type="xsd:integer"/>
...
<sequence>
  <assign>
    <copy>
      <from><literal>0</literal></from>
      <to variable="[counter]"/>
              </copy>
  </assign>
  <while>
    <condition>[p] and $[counter] &lt; [LoopMaximum]</condition>
    <sequence>
      [G]
      <assign>
        <copy>
          <from expression="$[counter]+1"/>
          <to variable="[counter]" />
        </copy>
      </assign>
    </sequence>
  </while>
</sequence>
```

(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

A standard *loop* with testTime=“After” and a `LoopMaximum` attribute maps as follows:



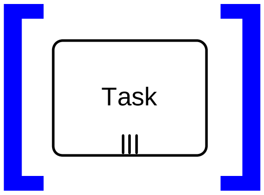
=

```
<variable name="[counter]" type="xsd:integer"/>
...
<sequence>
 <assign>
   <copy>
     <from><literal>0</literal></from>
     <to variable="[counter]"/>
              </copy>
 </assign>
 <repeatUntil>
   <sequence>
     [G]
     <assign>
       <copy>
         <from expression="$[counter]+1"/>
         <to variable="[counter]" />
       </copy>
     </assign>
   </sequence>
   <condition>[not p] or $[counter] &gt; [LoopMaximum]</condition>
 </repeatUntil>
</sequence>
```

(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

Multi-Instance Activities

A BPMN Multi-Instance Task with a `multiInstanceFlowCondition` of “All” is mapped to WS-BPEL as follows:



=

```
<variable name="[counter]" type="xsd:integer"/>
...
<forEach counterName="[counter]" parallel="[isSequential? 'no':'yes']">
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>[condition]</finalCounterValue>
  <scope>
    [Task]
  </scope>
</forEach>
```

(The notation `[counter]` denotes the unique name of a variable used to hold the counter value; the actual name is immaterial.)

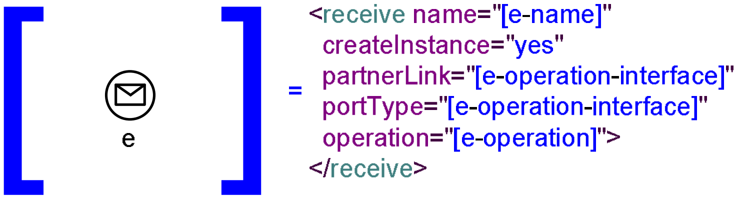
15.1.3. Events

Start Event Mappings

The following sections detail the mapping of Start Events to WS-BPEL.

Message Start Events

A Message Start Event is mapped to WS-BPEL as shown in the following figure:



```
<receive name="[e-name]"
  createInstance="yes"
  partnerLink="[e-operation-interface]"
  portType="[e-operation-interface]"
  operation="[e-operation]">
</receive>
```

The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the Message Start Event.

Error Start Events

An Error Start Event can only occur in Event Sub-Processes. This mapping is described on page 453.

Compensation Start Events

A Compensation Start Event can only occur in Event Sub-Processes. This mapping is described page 453.

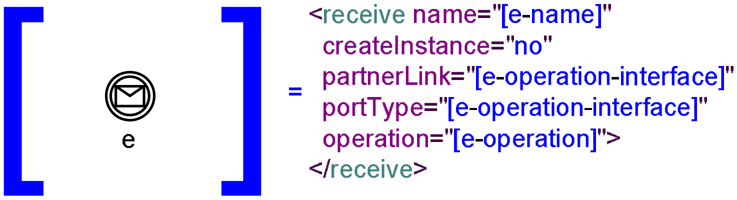
Intermediate Event Mappings (Non-boundary)

The following sections detail the mapping of intermediate non-boundary Events to WS-BPEL.

Message Intermediate Events (Non-boundary)

A Message Intermediate Event can either be used in normal control flow, similar to a Send or Receive Task (for *throw* or *catch* Message Intermediate Events, respectively), or it can be used in an Event Gateway. The latter is described in more detail in Section 15.1.4.

The following figure describes the mapping of Message Intermediate Events to WS-BPEL:



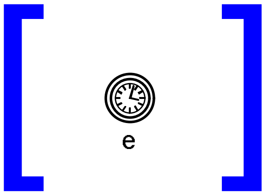
```
<receive name="[e-name]"
  createInstance="no"
  partnerLink="[e-operation-interface]"
  portType="[e-operation-interface]"
  operation="[e-operation]">
</receive>
```

The partner link associated with the WS-BPEL receive is derived from the interface referenced by the operation of the Message Intermediate Event.

Timer Intermediate Events (Non-boundary)

A Timer Intermediate Event can either be used in normal control flow, or it can be used in an Event Gateway. The latter is described in more detail in Section 15.1.4.

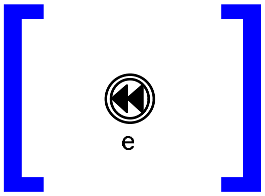
The following figure describes the mapping of a Timer Intermediate Event to WS-BPEL – note that one o the mappings shown is chosen depending on whether the Timer Event's TimeCycle or TimeDate attribute is used:



```
= <wait name="[e-name]" for="[e-TimeCycle]"/>
                         or
  <wait name="[e-name]" until="[e-TimeDate]"/>
```

Compensation Intermediate Events (Non-boundary)

A Compensation Intermediate Event with its waitForCompletion property set to *true*, that is used within an Event Sub-Process triggered through an *error* or through *compensation*, is mapped to WS-BPEL as follows:



```
= <compensate/>
                         or
  <compensateScope target="[referencedActivity]"/>
```

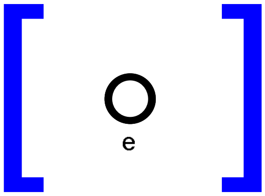
The first mapping is used if the Compensation Event does not reference an Activity, the second mapping is used otherwise.

End Event Mappings

The following sections detail the mapping of End Events to WS-BPEL.

None End Events

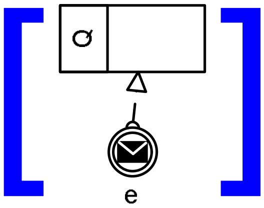
A "none" End Event marking the end of a Process is mapped to WS-BPEL as shown in the following figure:



```
= <empty name="[e-name]">
  </empty>
```

Message End Events

A Message Start Event is mapped to WS-BPEL as shown in the following figure:

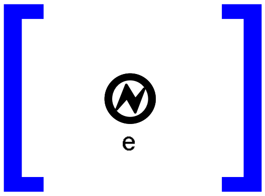


```
  <invoke name="[e-name]"
    partnerLink="[Q, e-operation-interface]"
=   portType="[e-operation-interface]"
    operation="[e-operation]">
  </invoke>
```

The partner link associated with the WS-BPEL invoke is derived from both the participant Q that the Message Intermediate Event is connected to by Message Flow, and from the interface referenced by the operation of the Message Intermediate Event.

Error End Events

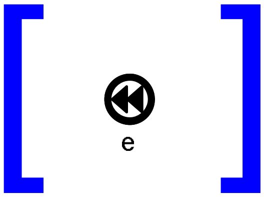
An Error End Event is mapped to WS-BPEL as shown in the following figure:



```
   <throw faultName="[e-name]">
=  </throw>
```

Compensation End Events

A Compensation End Event with its waitForCompletion property set to *true*, that is used within an Event Sub-Process triggered through an *error* or through *compensation*, is mapped to WS-BPEL as follows:

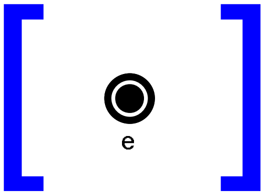


```
   <compensate/>
=                             or
   <compensateScope target="[referencedActivity]"/>
```

The first mapping is used if the Compensation Event does not reference an Activity, the second mapping is used otherwise.

Terminate End Events

A Terminate End Event is mapped to WS-BPEL as shown in the following figure:

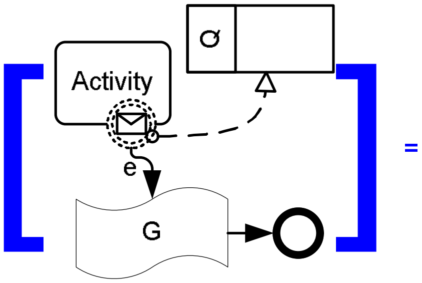


```
   <exit>
=  </exit>
```

Boundary Intermediate Events

Message Boundary Events

A BPMN Activity with a non-interrupting Message boundary Event is mapped to a WS-BPEL scope with an event handler as follows:



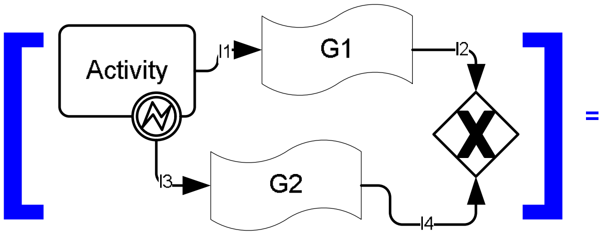
```
<scope>
  <eventHandlers>
    <onEvent partnerLink="[Q, e-operation-interface]"
        operation="[e-operation]">
      <scope>[G]</scope>
    </onEvent>
  </eventHandlers>
  [Activity]
</scope>
```

The partner link associated with the WS-BPEL onEvent is derived from the interface referenced by the operation of the boundary Message Event.

The same mapping applies to a non-interrupting boundary Timer Event, using a WS-BPEL onAlarm handler instead.

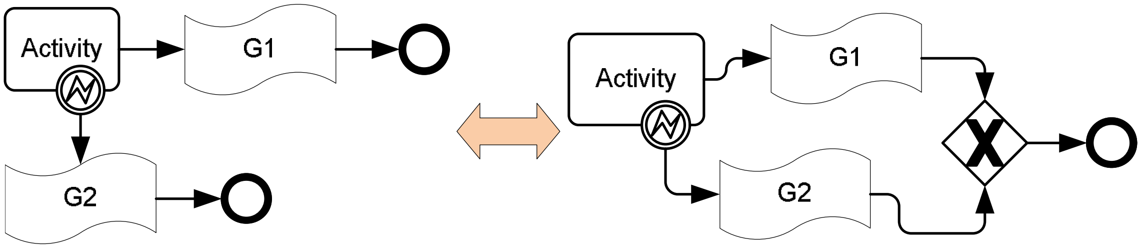
Error Boundary Events

A BPMN Activity with a boundary Error Event according to the following pattern is mapped as shown:



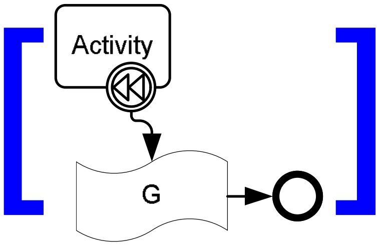
```
<flow>
  <links>
    <link name="[l1]"/>
    ...
    <link name="[l4]"/>
  </links>
  <scope>
    <sources><source linkName="[l1]"/></sources>
    <faultHandlers>
      <catch faultName="[e-error]">
        <empty>
          <sources><source linkName="[l3]"/></sources>
        </empty>
      </catch>
    </faultHandlers>
    [Activity]
  </scope>
  <flow>
    <targets><target linkName="[l1]"/></targets>
    <sources><source linkName="[l2]"/></sources>
    [G1]
  </flow>
  <flow>
    <targets><target linkName="[l3]"/></targets>
    <sources><source linkName="[l4]"/></sources>
    [G2]
  </flow>
  <empty>
    <sources><source linkName="[l2]"/>
    <source linkName="[l4]"/></sources>
  </empty>
</flow>
```

Note that the case where the error handling path doesn't join the main control flow again, is still mapped using this pattern, by applying the following model equivalence:



Compensation Boundary Events

A BPMN Activity with a *boundary* Compensation Event is similarly mapped as shown:



=

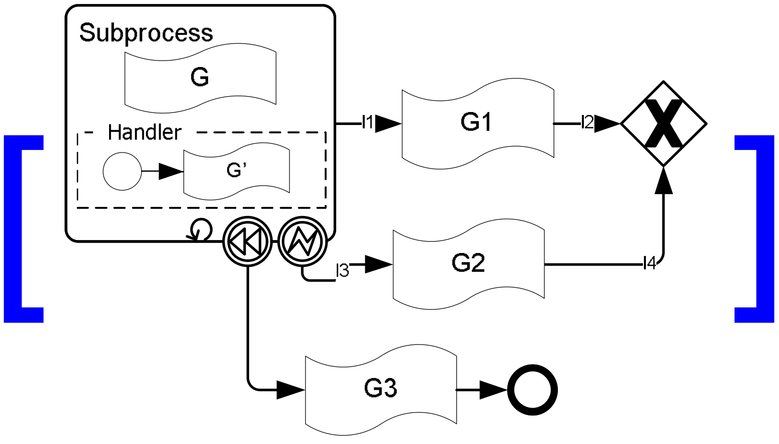
```
<scope name="[Activity-name]">
  <compensationHandler>
    [G]
  </compensationHandler>
  [Activity]
</scope>
```

Multiple Boundary Events, and Boundary Events with Loops

If there are multiple boundary Events for an Activity, their WS-BPEL mappings are super-imposed on the single WS-BPEL scope wrapping the mapping of the Activity.

When the Activity is a standard *loop* or a *multi-instance* and has one or more boundary Events, the WS-BPEL *loop* resulting from mapping the BPMN *loop* is nested inside the WS-BPEL scope resulting from mapping the BPMN boundary Events.

The following example shows that mapping for a Sub-Process with a nested Event Sub-Process that has a standard *loop* with TestTime="Before", an boundary Error Intermediate Event, and a boundary Compensation Intermediate Event.



=

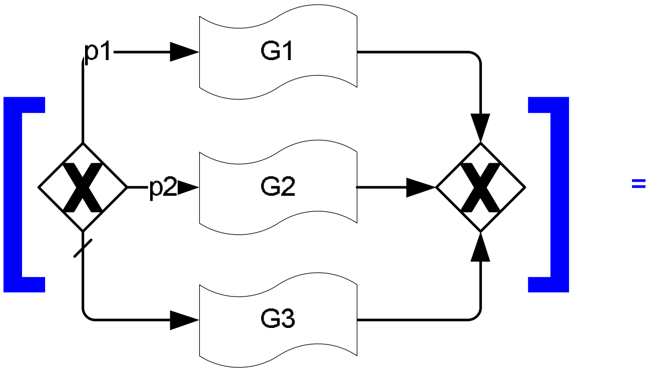
```
<flow>
  <links>
    <link name="[l1]"/>
    ...
    <link name="[l4]"/>
  </links>
  <scope>
    <sources><source linkName="[l1]"/></sources>
    <faultHandlers>
      <catch faultName="[e-error]">
        <empty>
          <sources><source linkName="[l3]"/></sources
        </empty>
      </catch>
    </faultHandlers>
    <compensationHandler>
      [G3]
    </compensationHandler>
    <while>
      <condition>[p]</condition>
      <scope>
        [Handler]
        [G]
      </scope>
    </while>
  </scope>
  <flow>
    <targets><target linkName="[l1]"/></targets>
    <sources><source linkName="[l2]"/></sources>
    [G1]
  </flow>
  <flow>
    <targets><target linkName="[l3]"/></targets>
    <sources><source linkName="[l4]"/></sources>
    [G2]
  </flow>
  <empty>
    <sources><source linkName="[l2]"/>
    <source linkName="[l4]"/></sources>
  </empty>
</flow>
```

15.1.4. Gateways and Sequence Flow

The mapping of BPMN Gateways and Sequence Flow is described using BPMN blocks following particular patterns.

Exclusive (Data-based) Decision Pattern

An exclusive data-based decision is mapped as follows:

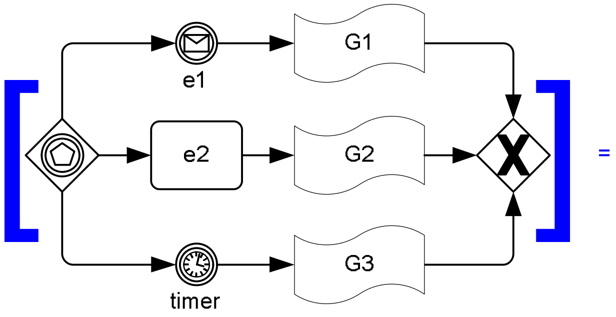


```
<if><condition>[p1]</condition>
  [G1]
<elseif><condition>[p2]</condition>
  [G2]
</elseif>
<else>
  [G3]
</else>
</if>
```

While this figure shows three branches, the pattern is generalized to *n* branches in an obvious manner.

Exclusive (Event-based) Decision Pattern

An Event Gateway is mapped as follows:

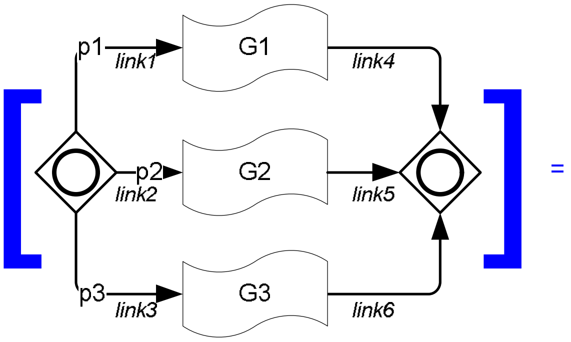


```
<pick createInstance="[instantiate? 'yes':'no']">
  <onMessage partnerLink="[e1-operation-interface]"
       operation="[e1-operation]">
    [G1]
  </onMessage>
  <onMessage partnerLink="[e2-operation-interface]"
       operation="[e2-operation]">
    [G2]
  </onMessage>
  <onAlarm>
    [timer-spec]
    [G3]
  </onAlarm>
</pick>
```

While this figure shows three branches with one Message Intermediate Event, one Receive Task and one Timer Intermediate Event, the pattern is generalized to *n* branches with any combination of the former in an obvious manner. The handling of *Participants* (BPEL partnerLinks), Event (operation) and timer details is as specified for Message Intermediate Events, Receive Tasks and Timer Intermediate Events, respectively. The data flow and associated variables (not shown) are handled as for Receive Tasks/Message Intermediate Events.

Inclusive Decision Pattern

An inclusive decision pattern without an otherwise gate is mapped as follows:



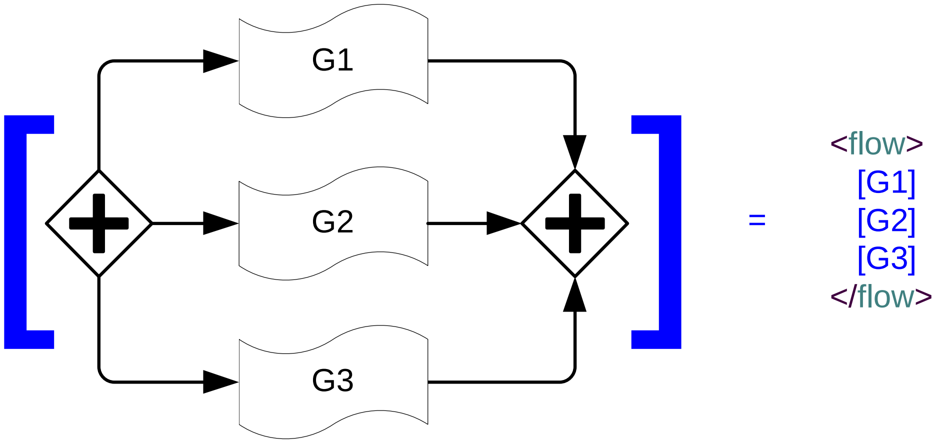
```
<flow>
  <links>
    <link name="[link1]"/>
    ...
    <link name="[link6]"/>
  </links>

  <empty>
    <sources>
      <source linkName="[link1]">
        <transitionCondition>[p1])</transitionCondition>
      </source>
      <source linkName="[link2]">
        <transitionCondition>[p2])</transitionCondition>
      </source>
      <source linkName="[link3]">
        <transitionCondition>[p3])</transitionCondition>
      </source>
    </sources>
  </empty>

  <flow>
    <targets><target linkName="[link1]"/></targets>
    <sources><source linkName="[link4]"/></sources>
    [G1]
  </flow>

  <flow>
    <targets><target linkName="[link2]"/></targets>
    <sources><source linkName="[link5]"/></sources>
    [G2]
  </flow>

  <flow>
    <targets><target linkName="[link3]"/></targets>
    <sources><source linkName="[link6]"/></sources>
    [G3]
  </flow>

  <empty>
    <targets>
      <target linkName="[link4]"/>
      <target linkName="[link5]"/>
      <target linkName="[link6]"/>
    </targets>
  </empty>
</flow>
```

While this figure shows three branches, the pattern is generalized to *n* branches in an obvious manner.

Note that link names in WS-BPEL must follow the rules of an XML NCName. Thus, the mapping of the BPMN Sequence Flow name attribute must appropriately canonicalize that name, possibly ensuring uniqueness, e.g., by appending a unique suffix. This is capture by the [linkName] notation.

Parallel Pattern

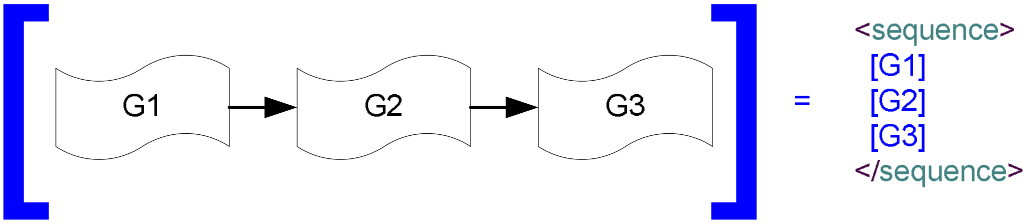
A parallel fork-join pattern is mapped as follows:



```
<flow>
  [G1]
  [G2]
  [G3]
</flow>
```

Sequence Pattern

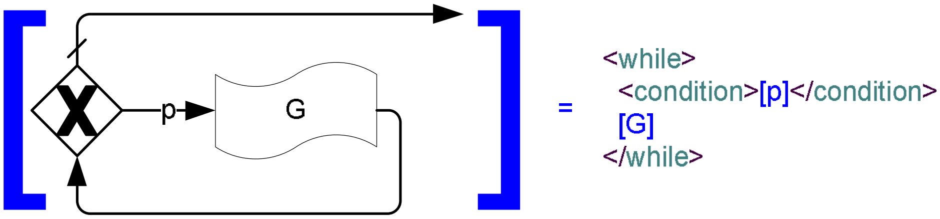
A BPMN block consisting of a series of Activities connected via (unconditional) Sequence Flow is mapped to a WS-BPEL `sequence`:



```
<sequence>
  [G1]
  [G2]
  [G3]
</sequence>
```

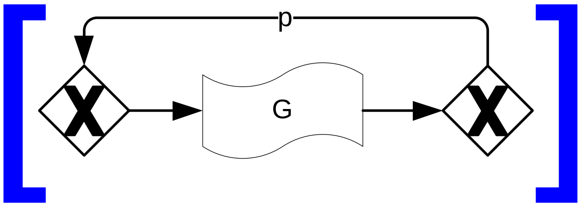
Structured Loop Patterns

A BPMN block consisting of a structured *loop* of the following pattern is mapped to a WS-BPEL `while`:



```
<while>
  <condition>[p]</condition>
  [G]
</while>
```

A BPMN block consisting of a structured *loop* of the following pattern is mapped to a WS-BPEL `repeatUntil`:



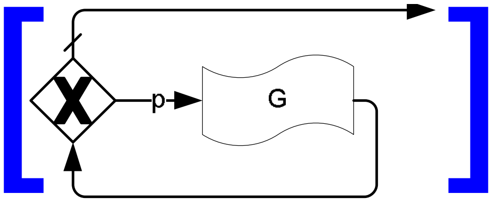
=

```
<repeatUntil>
  [G]
  <condition>[not p]</condition>
</repeatUntil>
```

Handling Loops in Sequence Flow

Loops are created when the flow of the Process moves from a downstream object to an upstream object. There are two types of *loops* that are WS-BPEL mappable: while *loops* and repeat *loops*.

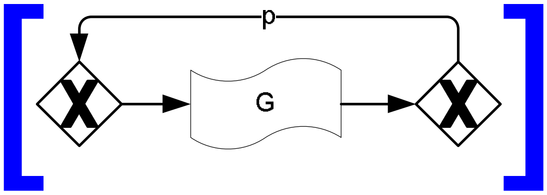
A while *loop* has the following structure in BPMN and is mapped as shown:



=

```
<while>
  <condition>[p]</condition>
  [G]
</while>
```

A repeat *loop* has the following structure in BPMN and is mapped as shown:



=

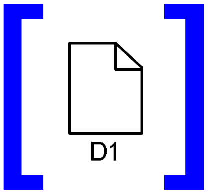
```
<repeatUntil>
  [G]
  <condition>[not p]</condition>
</repeatUntil>
```

15.1.5. Handling Data

Data Objects

BPMN Data Objects are mapped to WS-BPEL variables. The `itemDefinition` of the Data Object determines the XSD type of that variable.

Data Objects occur in the context of a Process or Sub-Process. For the associated WS-BPEL `process` or WS-BPEL `scope`, a variable is added for each Data Object in the corresponding WS-BPEL `variables` section, as follows:



=

```
<variable name="[D1-name]" type="[D1-structureDefinition]"/>
```

Properties

BPMN properties can be contained in a Process, Activity or an Event, here named the “container” of the property. A BPMN property is mapped to a WS-BPEL variable. Its name is derived from the name of its container and the name of the property. Note that in the case of different containers with the same name and a contained property of the same name, the mapping to WS-BPEL ensures the names of the associated WS-BPEL variables are unique. The itemDefinition of the property determines the XSD type of that variable.

A BPMN Process property is mapped to a WS-BPEL global variable. A BPMN Event property is mapped to a WS-BPEL variable contained in the WS-BPEL scope representing the immediately enclosing Sub-Process of the Event (or a global variable in case the Event is an immediate child of the Process). For a BPMN Activity property, two cases are distinguished: In case of a Sub-Process, the WS-BPEL variable is contained in the WS-BPEL scope representing the Sub-Process. For all other BPMN Activity properties, the WS-BPEL variable is contained in the WS-BPEL scope representing the immediately enclosing Sub-Process of the Activity (or a global variable in case the Activity is an immediate child of the Process).

```
[ <property id="P1-name"
   structureRef="P1-structureDefinition"/> ]
=
<variable name="[{container-name}.P1-name]"
    type="[P1-structureDefinition]"/>
```

Input and Output Sets

For a Send Task and a Service Task, the single input set is mapped to a WSDL message defining the input of the associated WS-BPEL activity. The inputs map to the message parts of the WSDL message. For a Receive Task and a Service Task, the single output set is mapped to a WSDL message defining the output of the associated WS-BPEL activity. The outputs map to the message parts of the WSDL message.

The structure of the WSDL message is defined by the itemDefinitions of the data inputs of the input set:

```
[ <inputSet name="iset">
   <dataInput name="input1">
     <structureDefinition structure="type1"/>
   </dataInput>
   ...
  </inputSet> ]
=
<wsdl:message name="[iset-name]">
  <part name="[input1-name]" type="[type1]"/>
  ...
</wsdl:message>
```

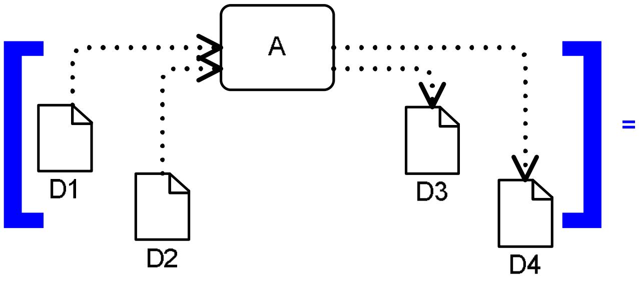
For the data outputs of the output set, the WSDL message looks as follows:

```
[ <outputSet name="oset">
   <dataOutput name="output1">
     <structureDefinition structure="type3"/>
   </dataOutput>
   ...
  </outputSet> ]
=
<wsdl:message name="[oset-name]">
  <part name="[output1-name]" type="[type3]"/>
  ...
</wsdl:message>
```

Data Associations

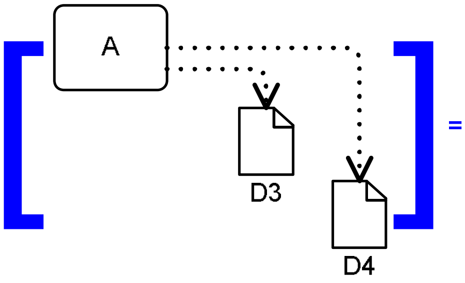
In this section, we assume that the input set of the Service Task has the same structure as its referenced input Message, and the output set of the Service Task has the same structure as its reference output Message. If this is not the case, assignments are needed, and the mapping is as described in the next section.

Data associations to and from a Service Task are mapped as follows:



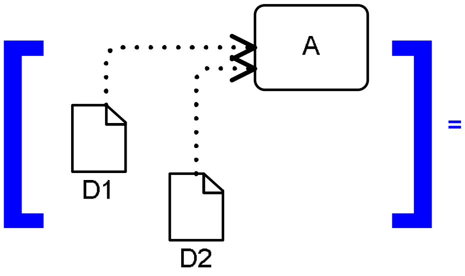
```
<invoke ...>
  <toParts>
    <toPart part="[dataInput1-name]"
      fromVariable="[D1-name]"/>
    <toPart part="[dataInput2-name]"
      fromVariable="[D2-name]"/>
  </toParts>
  <fromParts>
    <fromPart part="[dataOutput1-name]"
      fromVariable="[D3-name]"/>
    <formPart part="[dataOutput2-name]"
      fromVariable="[D4-name]"/>
  </fromParts>
</invoke>
```

Data associations from a Receive Task are mapped as follows:



```
<receive>
  <fromParts>
    <fromPart part="[dataOutput1-name]"
      fromVariable="[D3-name]"/>
    <formPart part="[dataOutput2-name]"
      fromVariable="[D4-name]"/>
  </fromParts>
</receive>
```

Data associations to a Send Task are mapped as follows:



```
<invoke>
  <toParts>
    <toPart part="[dataInput1-name]"
      fromVariable="[D1-name]"/>
    <toPart part="[dataInput2-name]"
      fromVariable="[D2-name]"/>
  </toParts>
</invoke>
```

Expressions

BPMN expressions specified using XPath (e.g., a condition expression of a Sequence Flow, or a timer cycle expression of a Timer Intermediate Event) are used as specified in BPMN, rewriting access to BPMN context to refer to the mapped BPEL context.

The BPMN XPath functions for accessing context from the perspective of the current Process are mapped to BPEL XPath functions for context access as shown in the following table. This is possible because the arguments must be literal strings.

Table 15-2 – Expressions mapping to WS-BPEL

BPMN context access	BPEL context access
getDataobject(dataObjectName)	$[dataObjectName]
getProcessProperty(propertyName)	$[{processName}.propertyName] where the right processName is statistically derived.
getActivityProperty(activityName, propertyName)	$[activityName.propertyName]
getEventProperty(eventName, propertyName)	$[eventName.propertyName]

Assignments

For a Service Tasks with assignments, the WS-BPEL mapping results in a sequence of an assign activity, an invoke activity and another assign activity. The first assign deals with creating the service request Message from the data inputs of the Task, the second assign deals with creating the data outputs of the Task from the service response Message.

15.2. Extended BPMN-BPEL Mapping

Additional sound BPMN Process models whose block hierarchy contains blocks that have not been addressed in the previous section can be mapped to WS-BPEL. For such BPMN Process models, in many cases there is no preferred single mapping of a particular block, but rather, multiple WS-BPEL patterns are possible to map that block to. Also, additional BPMN constructs can be mapped by using capabilities not available at the time of producing this specification, such as the upcoming OASIS BPEL4People standard to map BPMN User Tasks, or other WS-BPEL extensions.

Rather than describing or even mandating the mapping of such BPMN blocks, this specification allows for a semantic mapping of a BPMN Process model to an executable WS-BPEL process: The observable behavior of the target WS-BPEL process MUST match the operational semantics of the mapped BPMN Process. Also, the mappings described in Section 15.1 SHOULD be used where applicable.

15.2.1. End Events

End Events can be combined with other BPMN objects to complete the merging or joining of the paths of a WSBPEL structured element (see Figure 7-3).

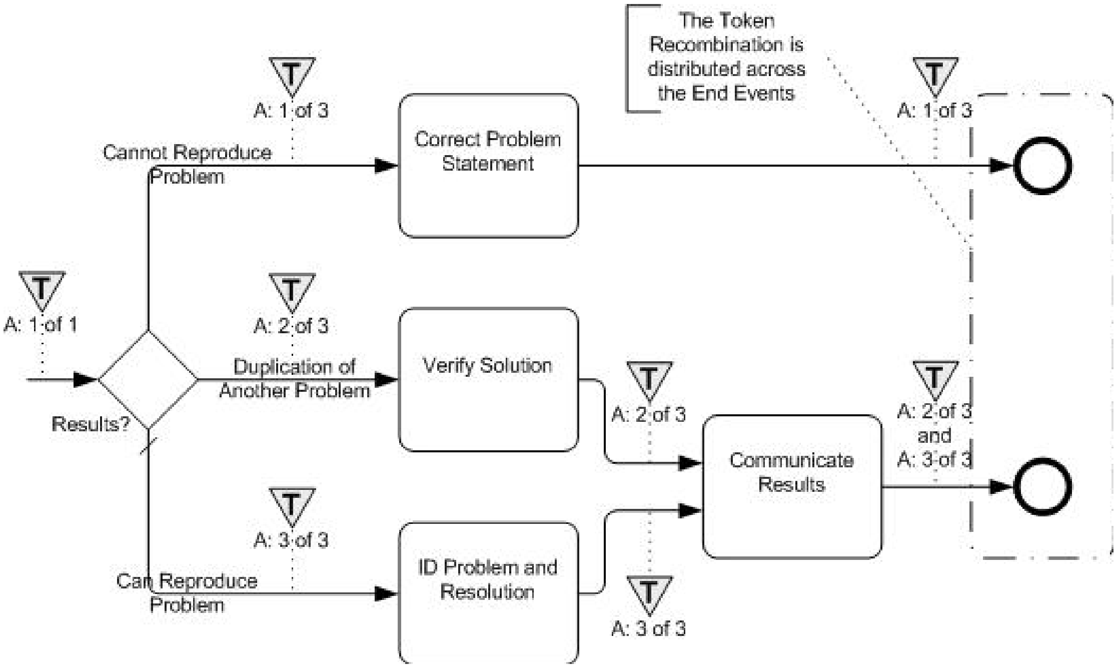


Figure 15-2 – An example of distributed *token* recombination

15.2.2. Loop/Switch Combinations From a Gateway

This type of *loop* is created by a Gateway that has three or more *outgoing* Sequence Flow. One Sequence Flow *loops* back *upstream* while the others continue *downstream* (see Figure 15-3). Note that there might be intervening Activities prior to when the Sequence Flow *loops* back *upstream.*

- This maps to both a WSBPEL *while* and a *switch*. Both activities will be placed within a *sequence*, with the *while* preceding the *switch*.
- For the *while*:
  - The *Condition* for the Sequence Flow that *loops* back *upstream* will map to the *condition* of the *while.*
  - All the Activities that span the distance between where the *loop* starts and where it ends, will be mapped and placed within the Activity for the *while*, usually within a *sequence.*
- For the *switch*:
  - For each additional *outgoing* Sequence Flow there will be a *case* for the *switch.*

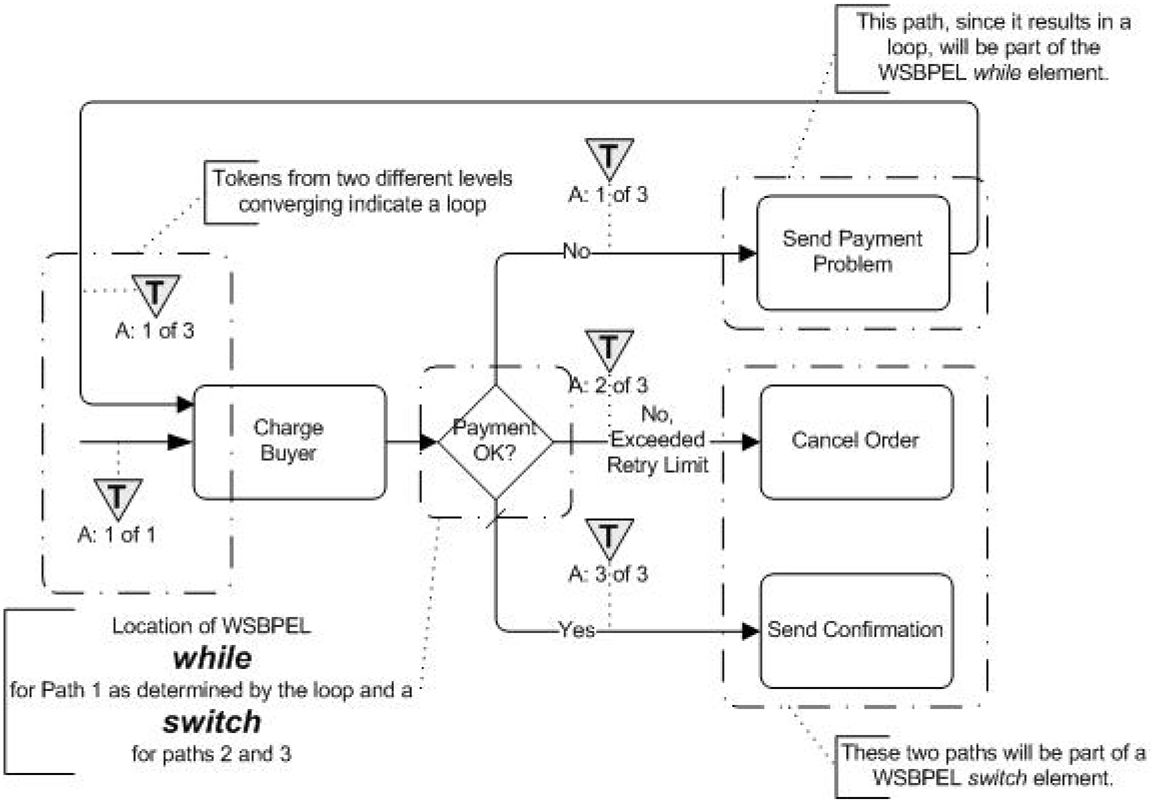

This path, since it results in a loop, will be part of the WSBPEL *while* element.
Tokens from two different levels converging indicate a loop
T
A: 1 of 3
No
Send Payment Problem
T
A: 1 of 3
T
A: 2 of 3
Charge Buyer
Payment OK?
No, Exceeded Retry Limit
Cancel Order
T
A: 1 of 1
T
A: 3 of 3
Yes
Send Confirmation
Location of WSBPEL **while** for Path 1 as determined by the loop and a **switch** for paths 2 and 3
These two paths will be part of a WSBPEL *switch* element.

Figure 15-3 – An example of a loop from a decision with more than two alternative paths

15.2.3. Interleaved Loops

This is a situation where there at least two *loops* involved and they are not nested (see Figure 15-4). Multiple looping situations can map, as described above, if they are in a sequence or are fully nested (e.g., one `while` inside another `while`). However, if the *loops* overlap in a non-nested fashion, as shown in Figure, then the structured element `while` cannot be used to handle the situation. Also, since a `flow` is acyclic, it cannot handle the behavior either.

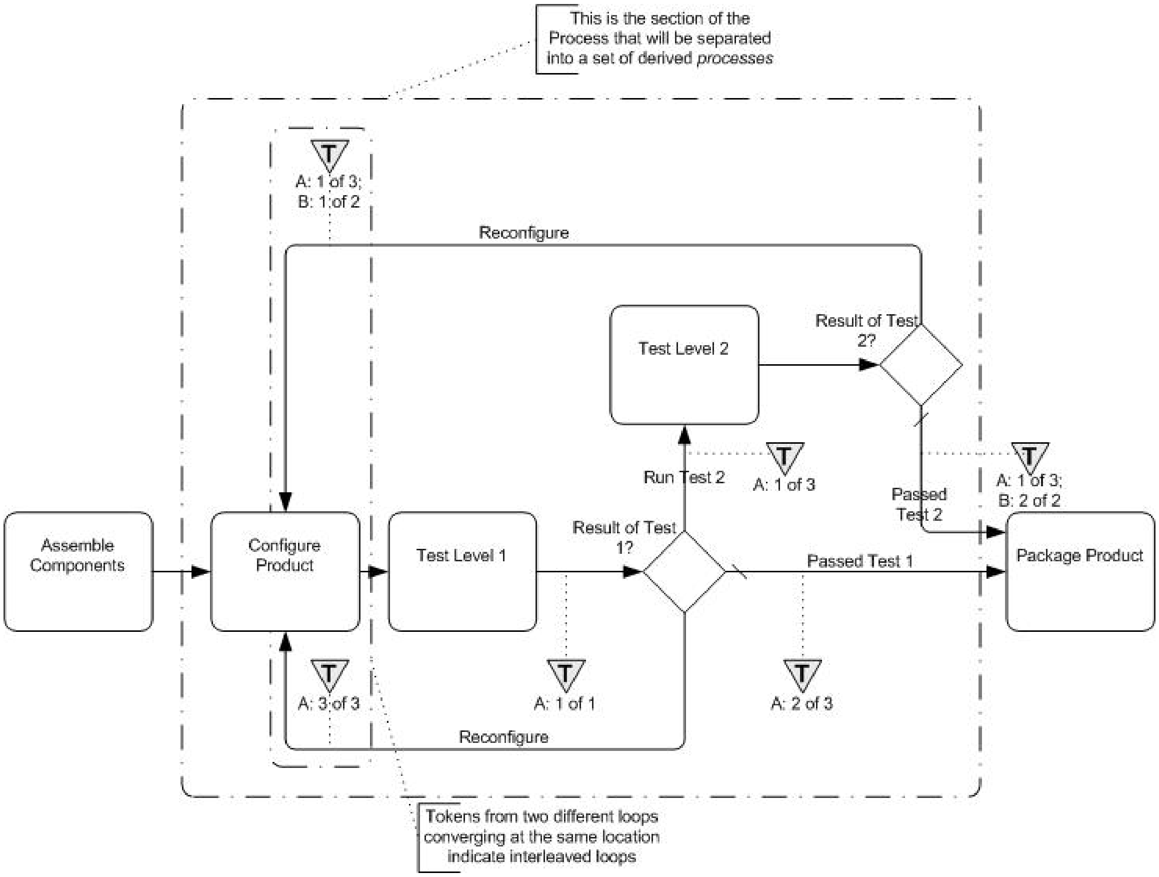


Figure 15-4 – An example of interleaved loops

To handle this type of behavior, parts of the WSBPEL *process* will have to be separated into one or more derived *processes* that are spawned from the main *process* and will also spawn or call each other (note that the examples below are using a spawning technique). Through this mechanism, the linear and structured elements of WSBPEL can provide the same behavior that is shown through a set of cycles in a single BPMN diagram. To do this:

- The looping section of the Process, where the *loops* first merge back (*upstream*) into the flow until all the paths have merged back to *Normal Flow*, shall be separated from the main WSBPEL *process* into a set of derived *processes* that will spawn each other until all the looping conditions are satisfied.
- The section of the *process* that is removed will be replaced by a (one-way) *invoke* to spawn the derived *process*, followed by a *receive* to accept the *message* that the looping sections have completed and the main *process* can continue (see Figure 15-5).
- The name of the *invoke* will be in the form of:
    - o "Spawn_[(loop target)activity.Name]_Derived_Process"
    - o The name of the *receive* will be in the form of:
    - o "[(loop target)activity.Name]_Derived_Process_Completed"

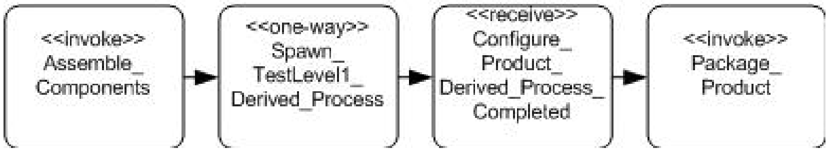

<<invoke>> Assemble_ Components
<<one-way>> Spawn_ TestLevel1_ Derived_Process
<<receive>> Configure_ Product_ Derived_Process_ Completed
<<invoke>> Package_ Product

Figure 15-5 – An example of the WSBPEL pattern for substituting for the derived *Process*

For each location in the Process where a Sequence Flow connects *upstream*, there will be a separate derived WSBPEL *process*.

- The name of the derived *process* will be in the form of:
    - o "[(loop target)activity.Name]_Derived_Process"
- All Gateways in this section will be mapped to *switch* elements, instead of *while* elements (see Figure below).
- Each time there is a Sequence Flow that *loops* back *upstream*, the Activity for the *switch case* will be a (one-way) *invoke* that will spawn the appropriate derived *process*, even if the *invoke* spawns the same *process* again.
- The name of the *invoke* will the same as the one describe above.
- At the end of the derived *process* a (one-way) *invoke* will be used to signal the main *process* that all the derived activities have completed and the main *process* can continue.
- The name of the *invoke* will be in the form of:
    - o "[(loop target)activity.Name]_Derived_Process_Completed"

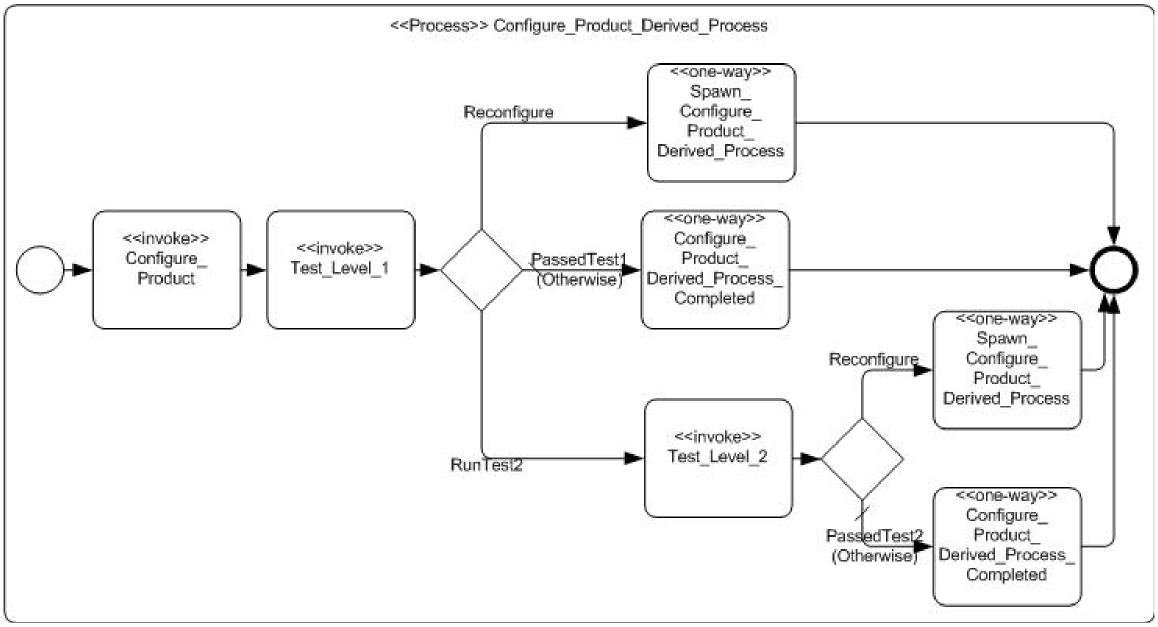

<<Process>> Configure_Product_Derived_Process
<<invoke>> Configure_ Product
<<invoke>> Test_Level_1
Reconfigure
<<one-way>> Spawn_ Configure_ Product_ Derived_Process
PassedTest1 (Otherwise)
<<one-way>> Configure_ Product_ Derived_Process_ Completed
RunTest2
<<invoke>> Test_Level_2
Reconfigure
<<one-way>> Spawn_ Configure_ Product_ Derived_Process
PassedTest2 (Otherwise)
<<one-way>> Configure_ Product_ Derived_Process_ Completed

Figure 15-6 – An example of a WSBPEL pattern for the derived *Process*

15.2.4. Infinite Loops

This type of *loop* is created by a Sequence Flow that *loops* back without an intervening Gateway to create alternative paths (see Figure 15-7). While this may be a modeling error most of the time, there may be situations where this type of *loop* is desired, especially if it is placed within a larger Activity that will eventually be interrupted.

- This will map to a *while* activity.
- The condition of the *while* will be set to an expression that will never evaluate to *true*, such as *condition* "1 = 0."
- All the activities that span the distance between where the *loop* starts and where it ends, will be mapped and placed within the activity for the *while*, usually within a *sequence*.

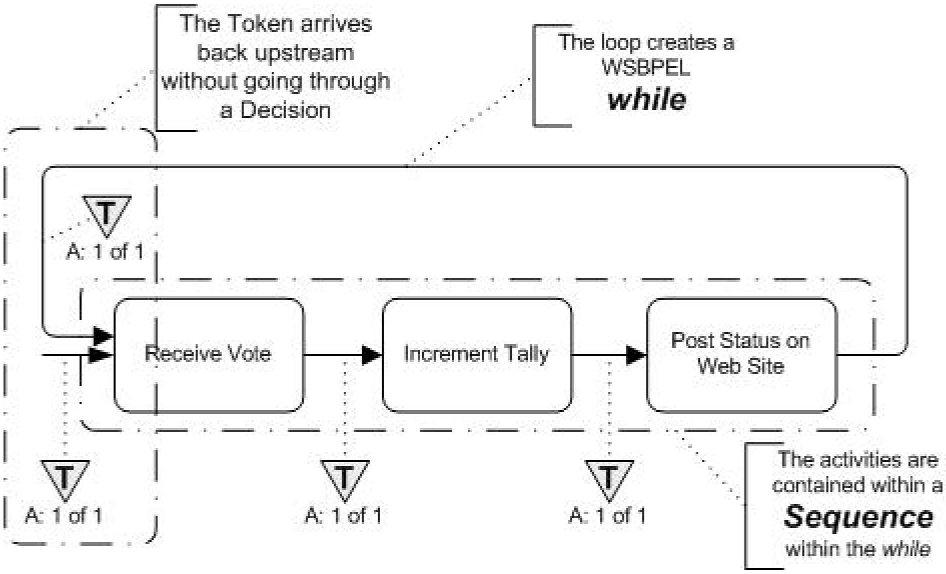

The Token arrives back upstream without going through a Decision
The loop creates a WSBPEL while
T
A: 1 of 1
Receive Vote
Increment Tally
Post Status on Web Site
T
A: 1 of 1
T
A: 1 of 1
T
A: 1 of 1
The activities are contained within a Sequence within the while

Figure 15-7 – An example – An infinite loop

15.2.5. BPMN Elements that Span Multiple WSBPEL Sub-Elements

Figure 15-8 below illustrates how BPMN objects may exist in two separate sub-elements of a WSBPEL structured element at the same time. Since BPMN allows free form connections of Activities and Sequence Flow, it is possible that two (or more) Sequence Flow will merge before all the Sequence Flow that map to a WSBPEL structure element have merged. The sub-elements of a WSBPEL structured elements are also self-contained and there is no cross sub-element flow. For example, the *cases* of a *switch* cannot interact; that is, they cannot share activities. Thus, one BPMN Activity will need to appear in two (or more) WSBPEL structured elements. There are two possible mechanisms to deal with the situation:

- First, the activities are simply duplicated in all appropriate WSBPEL elements.
- Second, the activities that need to be duplicated can be removed from the main Process and placed in a derived process that is called (*invoked*) from all locations in the WSBPEL elements as required.
  - o The name of the derived process will be in the form of:
    - ▪ "[(target)object.Name]_Derived_Process"

Figure 15-8 below displays this issue with an example. In that example, two Sequence Flow merge into the "Include History of Transactions" Task. However, the Decision that precedes the Task has three (3) alternatives. Thus, the Decision maps to a WSBPEL *switch* with three (3) *cases*. The three *cases* are not closed until the "Include Standard Text" Task, downstream. This means that the "Include History of Transactions" Task will actually appear in two (2) of the three (3) *cases* of the *switch*.

Note – the use of a WSBPEL *flow* will be able to handle the behavior without duplicating activities, but a *flow* will not always be available for use in these situations, particularly if a WSBPEL *pick* is required.

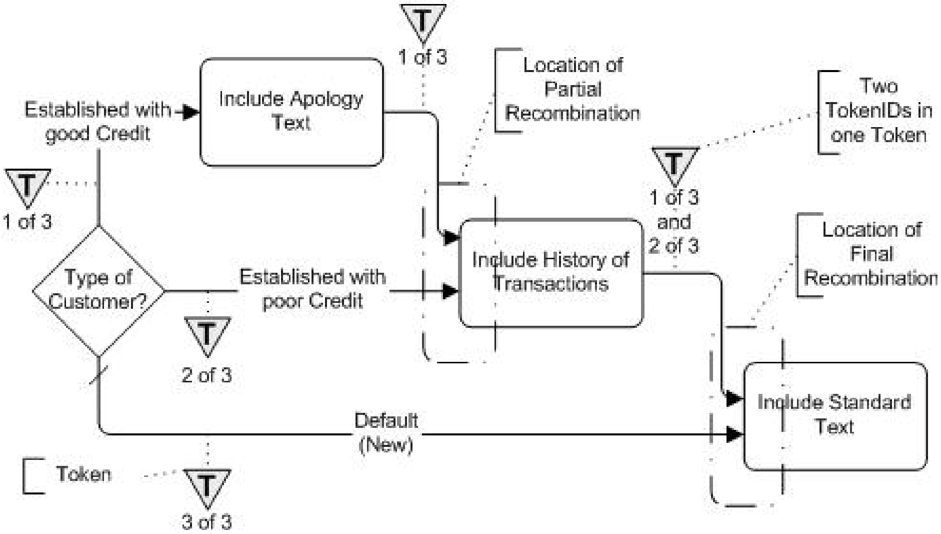

T
1 of 3
Location of Partial Recombination
Two TokenIDs in one Token
Established with good Credit
Include Apology Text
T
1 of 3
T
1 of 3 and 2 of 3
Location of Final Recombination
Type of Customer?
Established with poor Credit
Include History of Transactions
T
2 of 3
Include Standard Text
Default (New)
Token
T
3 of 3

Figure 15-8 – An example – Activity that spans two paths of a WSBPEL structured element

16. Exchange Formats

16.1. Interchanging Incomplete Models

In practice, it is common for models to be interchanged before they are complete. This occurs frequently when doing iterative modeling, where one user (such as a subject matter expert or business person) first defines a high-level model, and then passes it on to another user to be completed and refined.

Such "incomplete" models are ones in which all of the required attributes have not yet been filled in, or the cardinality lowerbound of attributes and associations has not been satisfied.

XMI allows for the interchange of such incomplete models. In BPMN, we extend this capability to interchange of XML files based on the BPMN XSD. In such XML files, implementers are expected to support this interchange by:

- Disregarding missing attributes that are marked as 'required' in the XSD.
- Reducing the lower bound of elements with 'minOccurs' greater than 0.

16.2. XSD

The BPMN 2.0 XSD for the interchange of semantic information can be found in OMG Document bmi/2009-05-05

The BPMN 2.0 XSD for the interchange of diagram information can be found in OMG Document bmi/2009-05-06

References within the BPMN XSD

All BPMN elements contain IDs and within the BPMN XSD, references to elements are expressed via these IDs. The XSD IDREF type is the traditional mechanism for referencing by IDs, however it can only reference an element within the same file. The BPMN XSD supports referencing by ID, across files, by utilizing QNames. A QName consists of two parts: an optional namespace prefix and a local part. When used to reference a BPMN element, the local part is expected to be the ID of the element.

For example, consider the following Process

```
<process name="Patient Handling" id="Patient_Handling_Process_ID1"> ... </process>
```

When this Process is referenced from another file, the reference would take the following form:

```
processRef="process_ns:Patient_Handling_Process_ID1"
```

where "process_ns" is the namespace prefix associated with the process namespace upon import, and "Patient_Handling_Process_ID1" is the value of the id attribute for the Process.

The BPMN XSD utilizes IDREFs wherever possible and resorts to QName only when references may span files. In both situations however, the reference is still based on IDs.

16.3. XMI

The BPMN 2.0 XMI for the interchange of semantic information can be found in OMG Document bmi/2009-05-04

The BPMN 2.0 XMI for the interchange of diagram information can be found in OMG Document bmi/2009-05-06

16.4. XSLT Transformation between XSD and XMI

The BPMN 2.0 XSLT for the transformation between XSD and XMI can be found in OMG Document bmi/2009-05-07

Annex A

(Informative)

Responses to RFP Requirements

The following tables provide a cross-reference between the requirements as stated in the Request for Proposals and the corresponding responses provided by this submission.

Mandatory Requirements

Table A-1 – Mandatory Requirements

Requirement	Resolution
Notation, Metamodel and Interchange Format Submissions shall define a single specification, entitled BPMN 2.0, that defines the notation, metamodel and interchange format. This specification will supersede BPDM 1.0 and BPMN 1.2.	This submission is predicated on the principal that BPMN requires a metamodel and interchange format whose constructs are clearly recognizable as BPMN elements. This requires that the correspondence of metamodel constructs to notional elements be as intuitive as possible. This submission's proposed BPMN 2.0 metamodel, therefore, is different than BPDM 1.0.
Extension of BPMN Notation Submissions shall define an extension of BPMN notation to address BPDM concepts.	The intent is to provide notation to address BPDM concepts.
Single, Consistent Language Submissions shall specify changes that are required to reconcile BPMN and BPDM to a single, consistent language.	Reconciliation of this submission's proposed metamodel with BPDM could be achieved via a metamodel-to-metamodel mapping; that is, via a mapping between the proposed metamodel and BPDM. Such a metamodel-to-metamodel mapping could supersede but be informed by BPDM 1.0's mapping to BPMN, which is a metamodel-to-notation mapping.
Model and Diagram Interchange Submissions shall provide the ability to use XMI to exchange business process models and their diagram layouts among process modeling tools.	TBD
Enhanced Notation Submissions shall define enhancements in BPMN's ability to model orchestrations and choreographies as stand-alone or integrated models.	TBD

Disposition of Outstanding Issues Submissions shall determine dispositions of outstanding issues not resolved by the respective finalization task forces for BPMN 1.2 and BPDM 1.0. The RFP response shall explain the reason that any outstanding issues are not addressed	TBD
MOF Compliance The resulting metamodel shall be MOF-compliant.	TBD

Optional Requirements

Table A-2 – Optional Requirements

Requirement	Resolution
6.6.1. Additional Normative or Non-Normative Mappings Proposals may provide additional mappings to recognized process definition languages, such as UML, SPEM, XPDL, ebBP, and WS-CDL	TBD
6.6.2. Additional perspectives Proposals may support the display and interchange of different perspectives on a model that allow a user to focus on specific concerns. The proposed perspectives shall be based on submitter experience with user needs.	TBD

Issues to be Discussed

Table A-3 – Issues to be Discussed

Issue to Discussed	**Resolution**
6.7.1. Relationships with related OMG specification activities Proposals shall discuss how the specifications relate to the specification development efforts currently under way as noted in Section 6.4.3	TBD
6.7.2. Consistency checks Proposals shall discuss how the specification supports checking and validating process models for consistency.	TBD
6.7.3. Terminology Submissions shall clarify the language and terms used in relation to models, diagrams, views and perspectives.	TBD

Changes from BPMN V1.2

There have been notational and technical changes to the BPMN specification.

The major notational changes include:

- The addition of a Choreography diagram
- The addition of a Conversation diagram
- Non-interrupting Events for a Process
- Event Sub-Processes for a Process

The major technical changes include:

- A formal metamodel as shown through the class diagram figures
- Interchange formats for semantic model interchange in both XMI and XSD
- Interchange formats for diagram interchange in both XMI and XSD
- XSLT transformations between the XMI and XSD formats

Annex B

(Non-Normative)

Diagram Interchange

This non-normative appendix explains the mechanism that was used to create BPMN 2.0's normative diagram interchange specification.[4] The mechanism is a generic approach to diagram interchange that can be applied to multiple languages, and that in this particular case was applied to BPMN 2.0. The mechanism has been submitted as a response to the OMG's Diagram Definition RFP. It is the intention of the BPMN 2.0 submitters that the BPMN 2.0 FTF will adjust BPMN 2.0's normative diagram interchange specification to align it with the outcome of the OMG's Diagram Definition specification process, and it is our intention that at that time the FTF will remove this appendix from the BPMN 2.0 specification.

Overview

The goal of the Diagram Interchange (DI) metamodel is to provide a way for BPMN to persist and interchange diagrams. Having common interchange format benefits tool interoperability, which is an ever increasing demand by end users. The DI metamodel, similar to the BPMN semantic metamodel, is defined as a MOF-based metamodel and hence its instances are serialized and interchanged with XMI.

Furthermore, a lot of the design decisions that characterize DI are motivated and influenced by experiences gained by working with similar technologies in the industry. Some of the major concerns in the industry for the adoption of any specification, but more critically for such a core one as DI, is its complexity, maintainability and scalability.

With regards to complexity concern, there are two major principles driving the design of DI. The first one is having a simple yet solid core, while allowing for variability using extensions. Obviously, there is no single diagram definition that suits the requirements of all possible domains and tools. However, it is certainly possible to define a minimal core that captures the main design pattern and make it extensible to address more specific requirements. Some typical extension mechanisms for metamodels include inheritance and redefinition. However, if not done very carefully (which is often the case), these extensions can easily lead to non-conforming extensions creeping in that would hurt the interchange and break the potential generality of diagramming tools. In addition, this would cause every domain or tool to have its own extended diagram metamodel and consequently XMI schema hindering reuse and interchange. For those reasons, the DI metamodel is kept closed for extension by inheritance. Instead, the metamodel focuses on defining a core design pattern for diagram persistence that is minimally constrained and allows for adding domain or tool specific extensions and/or constraints by referencing instances of another language called Diagram Definition (discussed in Chapter 13, as shown in Figure 16-1.

The second principle of managing complexity is the separation between the business data and its diagram data, or more idiomatically between the model and its view. The business data of BPMN is represented by its abstract syntax metamodel, where the diagram data needs to be captured in a separate metamodel (DI) that references the former as its context, as shown in Figure 16-1. There are several advantages to this design including: the ability

[4] The normative BPMN 2.0 diagram interchange specification has two parts. One part is Chapter 13 of this document. The other part is OMG document <omg document #>, which contains the diagram interchange schema

of both metamodels to evolve independently, the ability of BPMN elements to have multiple alternate notations, the ability of a BPMN element to be depicted more than once using the same notation and the ability of a notation to be defined using multiple diagram elements. This flexibility leads to more efficient and less bloated metamodel design for BPMN's diagram interchange.

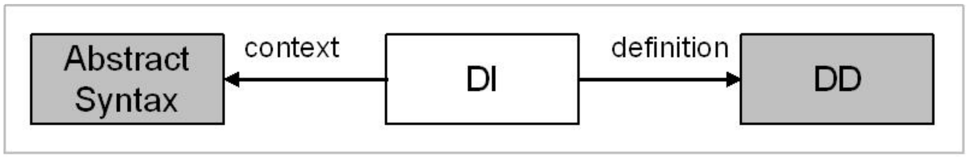


Figure 16-1 – The relationship between DI, DD and a BPMN's abstract-syntax metamodels

Regarding the maintainability concern, the fact that the DI metamodel is small and closed coupled with the built in separation of concern between the diagram interchange and its definition imply that the diagram interchange schema is less susceptible to change. This allows, for example, small extensions to be done to the diagram definition without affecting the diagram interchange, which can dramatically reduce the maintenance cost for tools. It also allows several unrelated extensions to be made without affecting their ability to coexist again reducing maintenance costs.

As for the scalability concern, the metamodel design focuses on eliminating redundancy as well as opting for alternatives that have more potential to scale better in realistic user setting. The scalability dimensions of importance here are memory footprint and change deltas. The details of the metamodel are given in the remainder of this chapter and wherever alternatives exist, a justification for the chosen alternative is given.

Metamodel Description

The underlying pattern of the DI metamodel is based on graph theory. The basic abstraction in DI is called a View, which describes an attributed graph element. A view owns a collection of name/value pairs representing its appearance attributes or styles. A View is further specialized into several kinds that correspond to the different components of a graph. One kind of view is Diagram, corresponding to the graph itself, which is the containment root for all other views in the diagram. The other kinds are Node and Connector. A Node corresponds to a graph node ad represents a child view that is contained by another view. A Connector corresponds to a graph connector and represents a relationship between two views (a source and a target). DI's view hierarchy is shown in Figure 16-2.

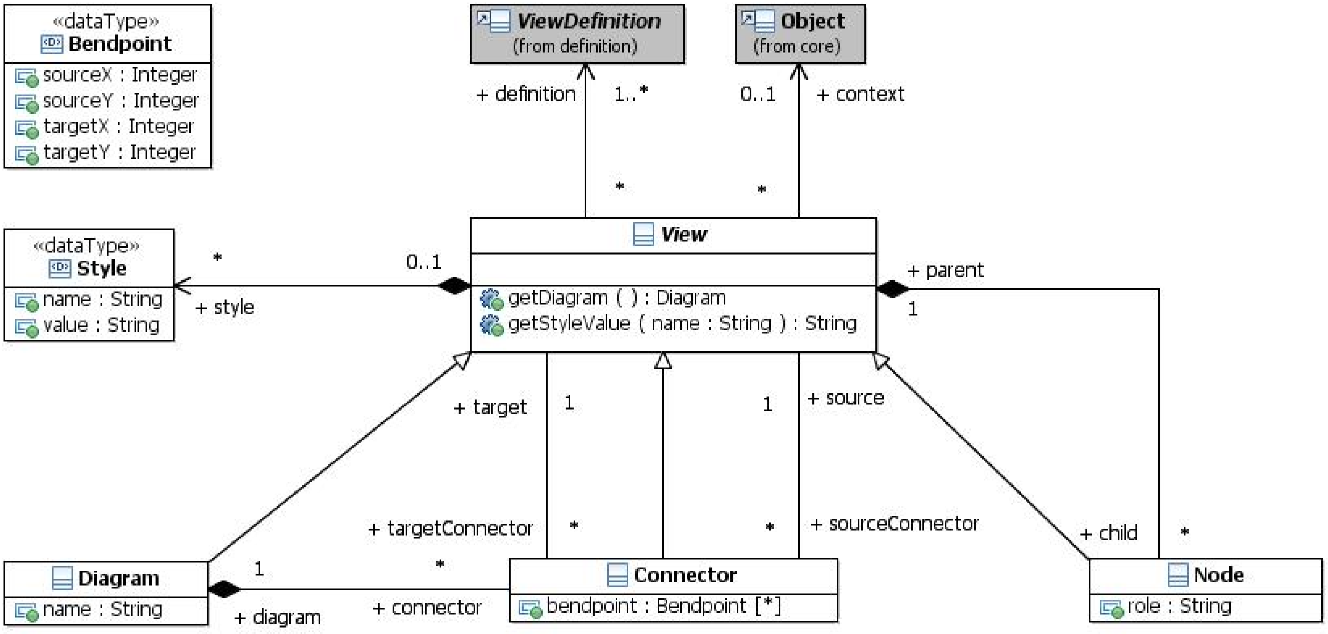


Figure 16-2 – Diagram Interchange (DI) Metamodel

As mentioned in the previous section, the DI metamodel is related to the abstract syntax metamodel and to the DD metamodel. The relationship to the abstract syntax metamodel is manifested in the view having an optional reference to MOF-based context object (which could be BPMN elements or others). When a view has a reference to a context object it is said to “visualize” this object. The relationship to the DD metamodel is manifested in the view having references to one or more view definitions. A view definition classifies a view, defines its proper form and specifies rules for its validation. These rules include definitions for the type of context object, the allowed styles and the allowed children of the view, in addition to arbitrary constraints. The first definition is the main definition that characterizes the view. The other optional definitions can be added by domain extensions and/or tool implementations to extend the definition of the view.

Class Description

View

A view is the main abstraction in the DI metamodel. It is also the building block of a diagram. It represents a unit of diagrammatic notation. A view may be purely notational, in which case it conveys information that is not in any other model. A view may also represent, by itself or with other views, the graphical notation of a context object from a MOF-based abstract syntax model (like BPMN models). View is an abstract metaclass that is further sub classed into three concrete kinds: Diagram, Node and Connector.

In addition to referencing a context object, a view contains a collection of styles (name/value pairs) representing its appearance properties (e.g. colors, line attributes, layout constraints...etc). It also contains a collection of nested child views, which add to its notation. A view also has two collections of references to source and target connectors. All these features give a view the flexibility to meet the syntactical requirement of a large set of domains, while still conforming to a common design pattern.

However, for a particular view instance, there must be a definition of its expected valid syntax. That is why a view references one or more view definitions that classify the view and specify its valid syntax. The first definition is the main one that defines the view. The other optional definitions allow for extending the valid syntax for domain and/or tool specific purposes. The conformance algorithm of a view to its view definitions is given as follows: (for more details about a view definition, see Section 13.2):

- A view’s context object has to conform to the context type specified by all its view definitions together.
- Each style owned by a view has to correspond to exactly one style definition contained by one of the view’s definitions. The correspondence is achieved when the name of a style matches the name of a style definition. Also the value of a style has to conform to the primitive type specified in its style definition.
- Each nested child view has to correspond to exactly one child definition contained by one of the view’s definitions. The correspondence is achieved when the role of a child matches the name of a child definition. The number of child views playing this role has to be compatible with the multiplicity specified by that child definition.
- All constraints contained by all view definitions referenced by a view have to be satisfied together on the view.

Furthermore, with regard to styles, DI specifies that a style can either be specified directly on the view or inherited from a parent view (if the style is defined as inherited in one of the style definitions owned by the view’s definitions). Therefore, the style value is calculated as follows:

- the view’s owned style value
- otherwise, the parent view’s style value (if the view has a parent and the style is inherited)
- otherwise, the style’s default value from the style definition
- otherwise, the style’s type’s natural default value

Properties

Table 16-4 – View attributes and model associations

Attribute Name	Description/Usage
definition: dd::ViewDefinition [1..*]	References a list of 1 or more references to DD view definitions that define the syntax of a view. At least one definition is required but other definitions are possible to allow tools to extend the valid syntax.
context: core::Object [0..1]	References an optional MOF-based object representing the context of the view
child: Node [0..*]	Contains a list of (nested) child nodes. The opposite end is Node::parent. The association defines the composite pattern for views.
sourceConnector: Connector [0..*]	References a list of outgoing connectors from the view. A source connector represents a relationship, in which the view is playing the source role. The opposite end is Connector::source.

targetConnector: Connector [0..*]	References a list of incoming connectors to the view. A target connector represents a relationship, in which the view is playing the target role. The opposite end is Connector::target.
styles: Style [0..*]	Contains a collection of styles (name/value pairs) providing override values for the view's appearance properties.

Constraints

- A view cannot have more than one style instance with the same name.
- A view has to conform to all its view definitions

Operations

- The query getDiagram() returns the view's diagram by walking the view's containment chain up to a diagram
- The query getStyleValue(name: String) gets the value of the style with the given name by applying the style calculation algorithm given above.

Diagram

A diagram is a special kind of view that has a name and designates the root of containment for all views in one diagram. A diagram directly contains all top level nodes, through the inherited 'child' association. It also directly contains all connectors in the same diagram regardless of the nesting level of their source and target views. This simplifies connector containment as it does not need to change in response to reconnections to different sources or targets.

Generalizations

- View

Properties

Table 16-5 – Diagram attributes and model associations

Attribute Name	Description/Usage
name: String	The name of the diagram
connector: Connector [0..*]	Contains a list of connectors in the diagram. Each connector's source and target views have to be nested in the same diagram.

Constraints

- A diagram can only be defined by `DiagramDefinitions`.

Node

A node is a special kind of view that can be nested (playing a child role) in some other view. It also represents a bounded area in the diagram that can be laid out (positioned and/or sized). The order of a node in its parent's child collection may or may not have an impact of how the node gets laid out. Nodes can represent notational idioms or play notational roles that are described in various graphical specifications. For example, they can represent "shapes" on diagrams, "compartments" on shapes, "labels" on connectors...etc.

Generalizations

- View

Properties

Table 16-6 – Node attributes and model associations

Attribute Name	**Description/Usage**
role: String	The role played by this node as a child in its parent.
parent: View	References the node's parent view that contains this node.

Constraints

- A node can only be defined by `NodeDefinitions`.
- The node's role must correspond to the name of exactly one of the child definitions contained by the view definitions defining the node's parent view.

Connector

A connector is a kind of view that connects two other views: a source view and a target view. A connector is rendered as a line going from the source to the target view. The line may be divided into segments by specifying bend points along its route. Bend points constrain the routing by forcing the connector's line to pass through them. A connector may own a collection of label nodes, through its inherited 'child' property. Labels are laid out relative to connector's line.

Generalizations

- View

Properties

Table 16-7 – Connector attributes and model associations

Attribute Name	**Description/Usage**

bendpoint: Bendpoint [0..*]	Contains a list of bendpoints for the connector. Each bendpoint specify an offset from the connector's source and target anchoring points.
source: View	References the connector's source view.
target: View	References the connector's source view.
diagram: Diagram	References the connector's diagram that owns the connector.

Constraints

- A connector can only be defined by ConnectorDefinitions.
- The connector's source view has to be nested in the same diagram as the connector.
- The connector's target view has to be nested in the same diagram as the connector.
- A connector cannot reference itself as a source or a target view.

Bendpoint

A bend point is a data type, which represents a point that a connector has to pass through in its route. A bend point is described by 2 offsets from the connector's source and target anchor points, as shown in Figure 16-3. Describing a bendpoint this way preserves its relative position when the connector's source and/or target change bounds.

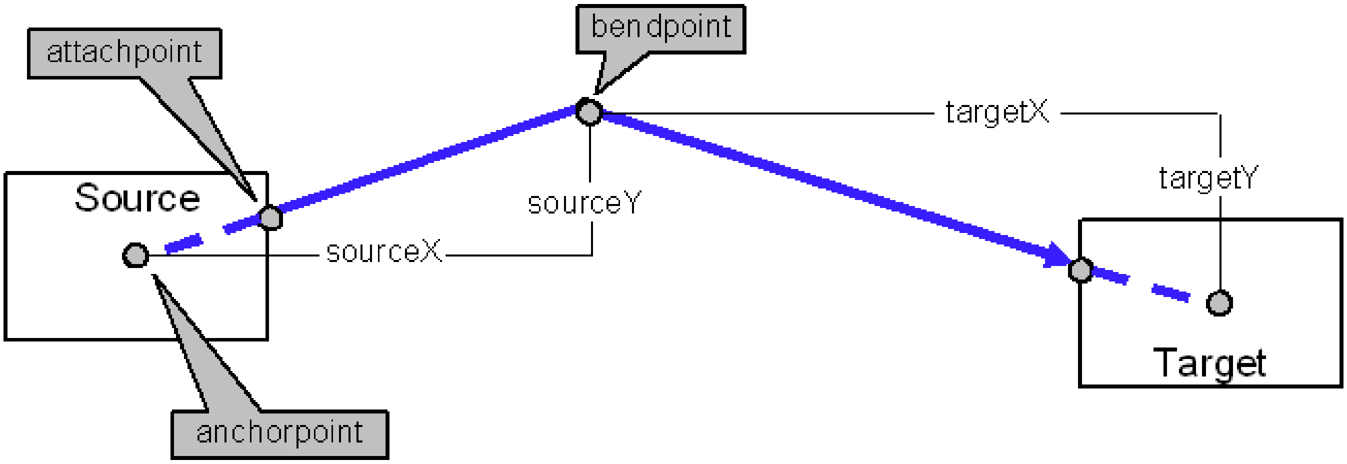


Figure 16-3 – Various points of a Connector

Properties

Table 16-8 – Bendpoint attributes and model associations

Attribute Name	Description/Usage
sourceX: Integer	The bendpoint's offset from the source anchor along the x-axis.
sourceY: Integer	The bendpoint's offset from the source anchor along the y-axis.
targetX: Integer	The bendpoint's offset from the target anchor along the x-axis.
targetY: Integer	The bendpoint's offset from the target anchor along the y-axis.

Style

A style is a data type consisting of a string name/value pair. It represents an appearance property for a view such as colors, line styles, layout constraints, drawing options...etc. The set of possible styles for a given view is given by the view's definition. Each style is defined by its name, type and default value. Instances of the style datatype represent a particular instance of a style with a given value.

Although the style's type may be an arbitrary data type, the style's value is always encoded as a string. For example if the style's type is a Boolean, the possible values would be the string literal "*true*" or "*false*".

Properties

Table 16-9 – Style attributes and model associations

Attribute Name	Description/Usage
name: String	The style's name string.
value: String	The style's value encoded as a string.

Constraints

- The style's string value is compatible with the style's defined type.

Diagram Definition

Overview

The previous chapter presented DI, a metamodel to create and persist diagrams. DI defines a schema for diagrams that is both flexible and domain independent. However, in order to interchange diagrams between domain specific tools, like BPMN tools, there must be a domain specific definition describing valid diagrams of that domain.

There are two approaches to having a domain specific diagram definition. One approach is to extend DI through MOF inheritance or package merge semantics to end up with a domain-specific DI. Although this approach benefits from having one diagram metamodel to use for each domain, it suffers from several practical drawbacks. First, it makes it more difficult to create or leverage general-purpose diagramming toolkits as it forces dealing with different possibly inconsistent diagram metamodels. Second, it reduces the potential of defining cross-domain hybrid diagrams which could undermine future integration efforts between domains. Third, it forces the resulting metamodel to have some parallel to the domain's abstract syntax metamodel resulting in a larger interchange schema that is also more susceptible to changes to the abstract syntax metamodel, increasing the maintenance cost.

The other approach to having a domain-specific diagram definition is to define it with a separate Diagram Definition (DD) language. The metamodel for DD is used to create M1 instances (model libraries representing diagram definitions) that are referenced by diagrams (DI instances). This architecture is adopted by this specification as it allows for several advantages. First, having a domain-independent DI means the interchange schema remains small and more stable (unaffected by changes to the abstract syntax), reducing maintenance costs. Second, having domain independent DI/DD metamodels allows for creating and/or leveraging general-purpose diagramming toolkits to define DSL modeling tools with, compressing time to market. Third, having a consistent DI/DD metamodels across domains eases the integration effort between specifications and/or tools, creating synergies and increasing business value. Fourth, separating DI and DD allows better separation of concerns. While DI is used to create and persist diagrams, DD is used to define the valid diagram syntax. This increases the flexibility as it allows diagram definitions to not be restricted by MOF/schema metamodel semantics. The relationship between these various metamodel is shown in Figure 16-1, above.

The main use case for associating diagrams with their definitions, expressed as DD instances, is diagram syntax validation. The DD metamodel defines and constrains various aspects of diagrams including composition rules, semantic references, and allowed styles. The other possible use cases for diagram definition are to help automate diagram creation with proper syntax and to help query and identify various parts of diagrams in a consistent way, which helps the genericity of diagramming tools.

DD instances are called diagram definition libraries as they are defined at the M1 level. The DD library for each standard domain, like BPMN, becomes part of the specification of that domain. Such a library is published with the specification and implemented by tool vendors. (The DD library for BPMN 2.0 is defined on page 398 of this document.) Hence, the standard DD libraries referenced from user diagrams do not need to be interchanged. On the other hand, if user diagrams reference non-standard (domain extension or tool specific) DD libraries and wish to interchange with other tools, those definition libraries need to also be available and recognized by those tools; otherwise, those definitions and their data would be ignored. In any case, the diagram data is still readable as it conforms to the same non-changing DI schema.

Metamodel Description

The DD metamodel, shown in Figure 16-4, provide definitions and constraints for the artifacts in the DI metamodel. That is why there is an obvious resemblance between the designs of both metamodels. At the core of the DD metamodel, there is the concept of a view definition. A view definition specifies various aspects of how a view should conform including the type and multiplicity of child views, the definition of the allowed styles, the type of the allowed context reference and other arbitrary constraints. A view definition is further sub classed by node definition, connector definition and diagram definition to match the different kinds of views in DI. View definitions are owned by a hierarchy of nested packages.

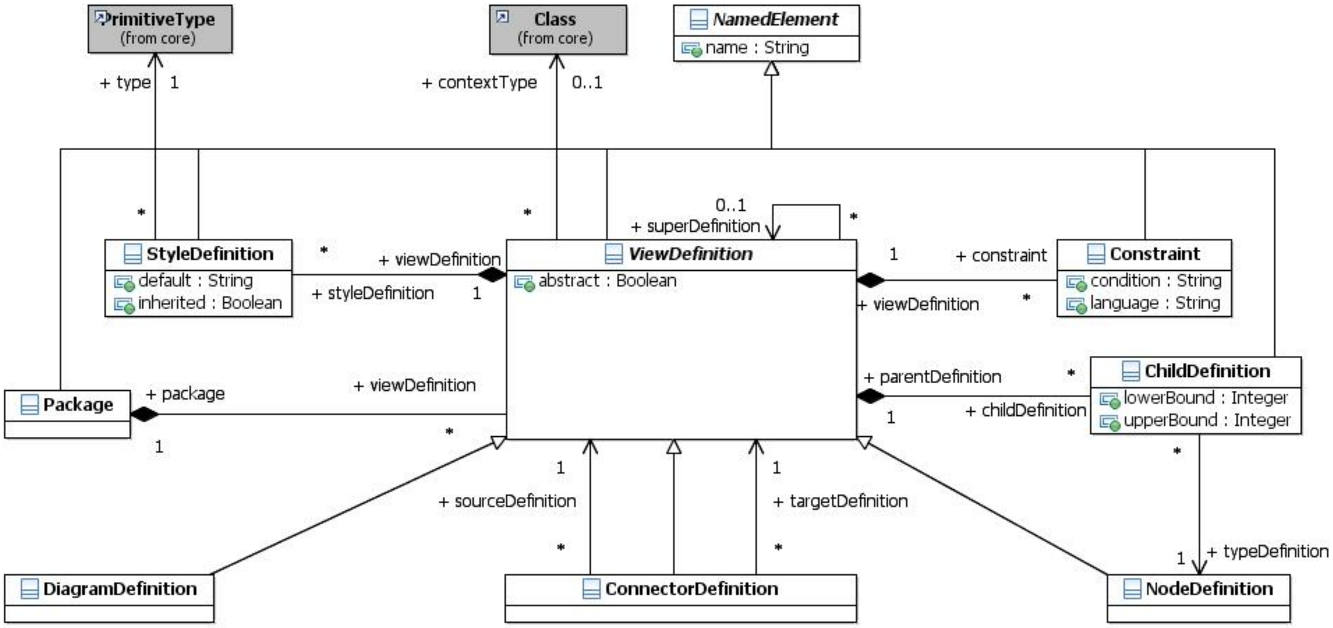


Figure 16-4 – Diagram Definition (DD) Metamodel

Class Descriptions

NamedElement

A named element is an element with a unique name within its siblings of the same type that are contained by the same container element if any. NamedElement is an abstract metaclass.

Properties

Table 16-10 – NamedElement attributes

Attribute Name	Description/Usage
name: String	The name of the `NamedElement`

Constraints

- The name of the element has to be unique within its siblings of the same type contained by the same container.

Package

A package is the root of containment in a DD model library that contains all view definitions in the library. It also represents a namespace for the library by having a namespace URI and a namespace prefix attributes, in addition to the name attribute inherited from NamedElement. The namespace URI uniquely identifies the package when referenced by other instance models. The namespace prefix is typically used as an alias to the URI to reduce its verboseness.

Generalizations

- `NamedElement`

Properties

Table 16-11 – Package model associations

Attribute Name	Description/Usage
viewDefinition: ViewDefinition [0..*]	Contains a list of view definitions in the DD instance library.

ViewDefinition

A view definition is the main abstraction in the DD metamodel. It is also the building block of a DD library. A view definition specifies the syntax rules for a view in DI. Various rules can be specified including the type and multiplicity of child views that can be composed in a view, the styles that can annotate a view, the type of the object that can be the context of a view, in addition to arbitrary constraints on a view. A view definition is abstract and s further sub classed by three concrete subclasses: diagram definition, node definition and connector definition.

The allowed context type of a view is specified by a reference to the allowed context metaclass on the view definition. When no context type is specified, a view cannot reference any context object. This is the case for purely notational views or views whose context is implied by other related views (ex. by the parent view for nodes or the source and target views for connectors). When the context type is specified, a view must reference a context object that conforms to (is instance of) this context type. For example, if a view definition references the type UML State, views conforming to this definition must reference a UML state object as a context.

The allowed styles of a view are specified by a collection of style definitions owned by the view definition. A conforming view can only be annotated by styles that conform to those style definitions. The conformance here is established when a style has the same name as well as a conforming value to one of the style definitions.

The allowed children of a view are specified by a collection of child definitions owned by the view definition. Each child definition specifies the allowed multiplicity (lower and upper bound) as well as the allowed type of a child. The type of a child definition is always a node definition. A conforming view owns children that conform to those child definitions. Child conformance is established when the child is defined by the child definition's type (or one of its subtypes) and respect its multiplicity (the number of children conforming to that child definition complies with its multiplicity).

A view definition can also specify arbitrary constraints on a view. Each constraint has a Boolean expression, expressed in some query language, in the context of the corresponding view's metaclass. A constraint can check any aspect of a view including its context object, its style values and its relationships to other views. A conforming view satisfies all of its definition's constraints.

In addition, view definitions can be defined as abstract or concrete and can be organized into inheritance hierarchies. Abstract definitions cannot be used to define views, while concrete definitions can be. To participate in an inheritance hierarchy, a view definition references another definition as its super definition. Only single inheritance is allowed for view definitions. The semantics of inheritance in this context is as follows:

- A sub view definition inherits a reference to a context type from its super definition chain. However, a sub view definition can refine the inherited context by specifying its own reference to a type that is either the same as or a subtype of the inherited context type.
- A sub view definition inherits the child definitions of its super definition chain. A sub definition can also provide its own child definitions that either add to or refine the inherited child definitions. If a new child definition has the same name as one of the inherited child definition, it is a refinement; otherwise it is an addition. A refining child definition specifies its own multiplicity and type, which is the same as or a subtype of the refined definition's type.
- A sub view definition inherits the style definitions of its super definition chain. A sub definition can also provide its own style definitions that are additions to the inherited ones. New style definitions must have different names than the inherited definitions.
- A sub view definition inherits the constraints of its super definition chain. A sub definition can also provide its own constraints that are additions to the inherited ones. New constraints must have different names than the inherited constraints.

Generalizations

- `NamedElement`

Properties

Table 16-12 – ViewDefinition attributes and model associations

Attribute Name	Description/Usage
abstract: Boolean [0..1]	A Boolean value that specifies whether the view definition is abstract. The default is 'false' meaning non-abstract (or concrete).
package: Package	References the package that contains this view definition.
superDefinition: ViewDefinition [0..1]	References an optional super (more general) view definition of this definition.

contextType: core::Class [0..1]	References an optional MOF-based metaclass representing the context type of this view definition. Views that conform to this view definition would reference a context object that is an instance of this metaclass.
childDefinition: NodeDefinition [0..*]	Contains a list of child definitions of this view definition. Child definitions define valid children of views conforming to this view definition.
styleDefinition: StyleDefinition [0..*]	Contains a list of style definitions of this view definition. Style definitions define valid styles of views conforming to this view definition.
constraint: Constraint [0..*]	Contains a list of constraints of this view definition. These constraints must be satisfied by views defined by this view definition.

Constraints

- View definition inheritance hierarchies must be directed and acyclic. A view definition cannot be both a transitively super and transitively sub definition of the same definition.
- A view definition's context type must either be the same as or a subtype of an inherited context type.
- A view definition must have a unique name in its containing package.

DiagramDefinition

A node definition is a concrete kind of view definition that defines the syntax rules of DI nodes.

Generalizations

- `ViewDefinition`

Constraints

- The super definition of a diagram definition must be of type diagram definition as well.

NodeDefinition

A diagram definition is a concrete kind of view definition that defines the syntax rules of DI diagrams.

Generalizations

- `ViewDefinition`

Constraints

- The super definition of a node definition must be of type node definition as well.

ConnectorDefinition

A connector definition is a concrete kind of view definition that defines the syntax rules of DI connectors. A connector definition references a source view definition and a target view definition. Those definitions define the valid source and target of a connector that conforms to this connector definition.

Generalizations

- `ViewDefinition`

Properties

Table 16-13 – NamedElement model associations

Attribute Name	Description/Usage
sourceDefinition: ViewDefinition	References a view definition that defines the valid source of a connector conforming to this connector definition.
targetDefinition: ViewDefinition	References a view definition that defines the valid target of a connector conforming to this connector definition.

ChildDefinition

A child definition specifies the conformance rules for child views that are contained by other views. Child definitions are owned by the view definitions of those containing views. In particular, a child definition specifies a name, inherited from NamedElement, representing a role played by conforming child views. It also specifies the valid multiplicity and type definition of those child views.

Generalizations

- `NamedElement`

Properties

Table 16-14 – ChildDefinition attributes and model associations

Attribute Name	Description/Usage
lowerBound: Integer [0..1]	The lower multiplicity of child views defined by this child definition.
upperBound: Integer [0..1]	The upper multiplicity of child views defined by this child definition.
typeDefinition: NodeDefinition	References a node definition that defines the child view.
parentDefinition: ViewDefinition	References a view definition that contains this child definition.

Constraints

- A child definition must have a unique name within its containing view definition.

StyleDefinition

A style definition is contained by a view definition. It specifies the conformance rules of a style that can annotate views that conform to that view definition. In particular, a style definition specifies a name, which needs to match a style's name to establish the correspondence between them. A style definition also specifies the type that a style value needs to conform to. This type is one of the primitive types (i.e. not structured type). A default value can also be specified.

Additionally, a style definition specifies whether a style is inherited or not. An inherited style allows views that are not annotated themselves with this style to "inherit" the style from the closest parent that is annotated with this style in their parent chain.

Properties

Table 16-15 – StyleDefinition attributes and model associations

Attribute Name	Description/Usage
default: String [0..1]	An optional default value of the style.
inherited: Boolean[0..1]	Specifies whether the style is inherited or not. The default is ‘false’, i.e. not inherited.
type: core::PrimitiveType	References the MOF-based primitive type of this style
viewDefinition: ViewDefinition	References a view definition that contains this style definition.

Constraint

A constraint represents an arbitrary condition expressed in some query language that must be satisfied by views conforming to the view definition that contains this constraint. The context type of a constraint corresponds to the view metaclass implied by the view definition that owns the constraint.

Generalizations

- `NamedElement`

Properties

Table 16-16 – Constraint attributes and model associations

Attribute Name	Description/Usage
condition: String	A condition that is specified in the context of a view metaclass that corresponds to the view definition that owns this constraint. For example, if the constraint is owned by a node definition, the condition’s context would be the metaclass Node from DI.
language: String [0..1]	The query language used to specify the condition. It is optional with a default of ‘OCL’.
viewDefinition: ViewDefinition	References a view definition that contains this constraint.

Annex C

(Informative)

Glossary

[The Glossary has been taken from the BPMN 1.1 specification and needs to be updated.]

A

Activity: An activity is a generic term for work that company or organization performs via business processes. An activity can be atomic or non-atomic (compound). The types of activities that are a part of a Process Model are: Process, Sub-Process, and Task.

AND-Join: (from the WfMC Glossary[5]) An AND-Join is a point in the Process where two or more parallel executing activities converge into a single common thread of Sequence Flow. See “Join.”

AND-Split: (from the WfMC Glossary[2]) An AND-Split is a point in the Process where a single thread of Sequence Flow splits into two or more threads which are executed in parallel within the Process, allowing multiple activities to be executed simultaneously. See “Fork.”

Artifact: An Artifact is a graphical object that provides supporting information about the Process or elements within the Process. However, it does not directly affect the flow of the Process. BPMN has standardized the shape of a Data Object. Other examples of Artifacts include critical success factors and milestones.

Association: An Association is a dotted graphical line that is used to associate information and Artifacts with Flow Objects. Text and graphical non-Flow Objects can be associated with the Flow Objects and Flow.

Atomic Activity: An atomic activity is an activity not broken down to a finer level of Process Model detail. It is a leaf in the tree-structure hierarchy of Process activities. Graphically it will appear as a Task in BPMN.

[5] The underlined terms in this definition were changed from the original definition. “Process” is used in place of “workflow.” “Sequence Flow” is used in place of “control.”

B

Business Analyst: A Business Analyst is an individual within an organization who defines, manages, or monitors Business Processes. They are usually distinguished from the IT specialists or programmers who implement the Business Process within a BPMS.

Business Process: TBD.

Business Process Management: Business Process Management (BPM) encompasses the discovery, design, and deployment of business processes. In addition, BPM includes the executive, administrative, and supervisory control of those processes[6].

BPM System: The technology that enables BPM.

C

Choreography: Choreography is an ordered sequence of B2B message exchanges.

Collaboration: Collaboration describes interactions between two or more PartnerEntities or PartnerRoles.

Collaboration Process: A Collaboration Process depicts the interactions between two or more business entities.

Collapsed Sub-Process: A Collapsed Sub-Process is a Sub-Process that hides its flow details. The Collapsed Sub-Process object uses a marker to distinguish it as a Sub-Process, rather than a Task. The marker is a small square with a plus sign (+) inside.

Compensation Flow: Compensation Flow is defines the set of activities that are performed during the roll-back of a transaction to compensate for activities that were performed during the Normal Flow of the Process.

Compensation can also be called from a Compensate End or Intermediate Event.

Compound Activity: A compound activity is an activity that has detail that is defined as a flow of other activities. It is a branch (or trunk) in the tree-structure hierarchy of Process activities. Graphically, it will appear as a Process or Sub-Process in BPMN.

[6] From “Business Process Management: the Third Wave,” by Howard Smith and Peter Fingar, pg 4. 2003, Meghan-Kiffer Press. ISBN 0-929652-33-9

Controlled Flow: Flow that proceeds from one Flow Object to another, via a Sequence Flow link, but is subject to either conditions or dependencies from other flow as defined by a Gateway.

Typically, this is seen as a Sequence flow between two activities, with a conditional indicator (mini-diamond) or a Sequence Flow connected to a Gateway.

D

Decision: Decisions are locations within a business process where the Sequence Flow can take two or more alternative paths. This is basically the “fork in the road” for a process. For a given performance (or instance) of the process, only one of the forks can be taken. A Decision is a type of Gateway. See “Or-Split.”

E

End Event: As the name implies, the End Event indicates where a process will end. In terms of Sequence Flow, the End Event ends the flow of the Process, and thus, will not have any outgoing Sequence Flow. An End Event can have a specific Result that will appear as a marker within the center of the End Event shape. End Event Results are Message, Error, Compensation, Link, and Multiple. The End Event shares the same basic shape of the Start Event and Intermediate Event, a circle, but is drawn with a thick single line

Event Context: An Event Context is the set of activities that can be interrupted by an exception (Intermediate Event).

This can be one activity or a group of activities in an expanded Sub-Process.

Exception: An Exception is an event that occurs during the performance of the process that causes Normal Flow of the process to be diverted exclusively from Normal Flow. Exceptions can be generated by a time out, fault, message, etc.

Exception Flow: Exception Flow is a set of Sequence Flow that originates from an Intermediate Event that is attached to the boundary of an activity. The Process will not traverse this flow unless an Exception occurs during the performance of that activity (through an Intermediate Event).

Expanded Sub-Process: An Expanded Sub-Process is a Sub-Process that exposes its flow detail within the context of its Parent Process. It will maintain its rounded rectangle shape, but will be enlarged to a size sufficient to display the Flow Objects within.

F

Flow: A Flow is a graphical line connecting two objects in a BPMN diagram. There are two types of Flow: Sequence Flow and Message Flow, each with their own line style. Flow is also used in a generic sense (and lowercase) to describe how *Tokens* will traverse Sequence Flow from the Start Event to an End Event.

Flow Object: A Flow Object is one of the set of following graphical objects: Events, Activities, and Gateways.

Fork: A fork is a point in the Process where a single flow is divided into two or more Flow. It is a mechanism that will allow activities to be performed concurrently, rather than sequentially. BPMN uses multiple outgoing Sequence Flow or an Parallel Gateway to perform a Fork. See “AND-Split.”

I

Intermediate Event: An Intermediate Event is an event that occurs after a Process has been started. It will affect the flow of the process, but will not start or (directly) terminate the process. An Intermediate Event will show where messages or delays are expected within the Process, disrupt the Normal Flow through exception handling, or show the extra flow required for compensating a transaction. The Intermediate Event shares the same basic shape of the Start Event and End Event, a circle, but is drawn with a thin double line.

J

Join: A Join is a point in the Process where two or more parallel Sequence Flow are combined into one Sequence Flow. BPMN uses an Parallel Gateway to perform a Join. See “AND-Join.”

L

Lane: An Lane is a sub-partition within a Pool and will extend the entire length of the Pool, either vertically or horizontally. Lanes are used to organize and categorize activities within a Pool. The meaning of the Lanes is up to the modeler.

M

Merge: A Merge is a point in the process where two or more alternative Sequence Flow are combined into one

Sequence Flow. BPMN uses multiple incoming Sequence Flow or an Exclusive Gateway to perform a Merge. See “OR-Join.”

Message: A Message is the object that is transmitted through a Message Flow. The Message will have an identity that can be used for alternative branching of a Process through the Event-Based Exclusive Gateway.

Message Flow: A Message Flow is a dashed line that is used to show the flow of messages between two entities that are prepared to send and receive them. In BPMN, two separate Pools in the Diagram will represent the two entities.

Normal Flow: Normal Flow is the flow that originates from a Start Event and continues through activities via alternative and parallel paths until it ends at an End Event.

O

OR-Join: (from the WfMC Glossary[7]) An OR-Join is a point in the Process where two or more alternative activity(s) Process branches re-converge to a single common activity as the next step within the Process. (As no parallel activity execution has occurred at the join point, no synchronization is required.) See “Merge.”

OR-Split: (from the WfMC Glossary[1]) An OR-Split is a point in the Process where a single thread of Sequence Flow makes a decision upon which branch to take when encountered with multiple alternative Process branches. See “Decision.”

P

Parent Process: A Parent Process is the Process that holds a Sub-Process within its boundaries.

Participant: A Participant is a Partner Entity (e.g., a company, company division, or a customer) or a Partner Role (e.g., a buyer or a seller), which controls or is responsible for a business process. If Pools are used, then a Participant would be represented by a Pool.

Pool: A Pool represents a Participant in a Process. It also acts as a “swimlane” and a graphical container for partitioning a set of activities from other Pools, usually in the context of B2B situations. It is a square-cornered rectangle that

[7] The underlined terms in this definition were changed from the original definition. “Process” is used in place of “workflow.” “Sequence Flow” is used in place of “control.”

is drawn with a solid single line. A Pool acts as the container for the Sequence Flow between activities. The Sequence Flow can cross the boundaries between Lanes of a Pool, but cannot cross the boundaries of a Pool. The interaction between Pools, e.g., in a B2B context, is shown through Message Flow.

Private Process: A private Process is internal to a specific organization and is the type of process that has been generally called a workflow or BPM Process. There are two (2) types of private Processes: executable and non-executable. A single executable private Process will map to a single BPEL document.

Process: A Process is any activity performed within a company or organization. In BPMN a Process is depicted as a network of Flow Objects, which are a set of other activities and the controls that sequence them.

Public Process: A Public Process represents the interactions between a private Business Process and another Process or Participant.

R

Result: A Result is consequence of reaching an End Event. Results can be of different types, including: Message, Error, Compensation, Link, and Multiple.

S

Sequence Flow: A Sequence Flow is a solid graphical line that is used to show the order that activities will be performed in a Process. Each Flow has only one source and only one target.

Start Event: A Start Event indicates where a particular Process will start. In terms of Sequence Flow, the Start Event starts the flow of the Process, and thus, will not have any incoming Sequence Flow. A Start Event can have a Trigger that indicates how the Process starts: Message, Timer, Rule, Link, or Multiple. The Start Event shares the same basic shape of the Intermediate Event and End Event, a circle, but is drawn with a single thin line

Sub-Process: A Sub-Process is Process that is included within another Process. The Sub-Process can be in a collapsed view that hides its details. A Sub-Process can be in an expanded view that shows its details within the view of the Process in which it is contained. A Sub-Process shares the same shape as the Task, which is a rectangle that has rounded corners.

Swimlane: A Swimlane is a graphical container for partitioning a set of activities from other activities. BPMN has two different types of Swimlanes. See “Pool” and “Lane.”

Task: A Task is an atomic activity that is included within a Process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user and/or an application are used to perform the Task when it is executed. A Task object shares the same shape as the Sub-Process, which is a rectangle that has rounded corners.

Token: A *Token* is a descriptive construct used to describe how the flow of a process will proceed at runtime. By tracking how the *Token* traverses the Flow Objects, gets diverted through alternative paths, and gets split into parallel paths, the normal Sequence Flow should be completely definable. A *Token* will have a unique identity that can be used to separate multiple *Tokens* that may exist because of concurrent process instances or the splitting of the *Token* for parallel processing within a single process instance.

Transaction: A Transaction is a set of coordinated activities carried out by independent, loosely-coupled systems in accordance with a contractually defined business relationship. This coordination leads to an agreed, consistent, and verifiable outcome across all participants.

Trigger: A Trigger is a mechanism that signals the start of a business process. Triggers are associated with a Start Events and Intermediate Events and can be of the type: Message, Timer, Rule, Link, and Multiple.

U

Uncontrolled Flow: Flow that proceeds, unrestricted, from one Flow Object to another, via a Sequence Flow link, without any dependencies on another flow or any conditional expressions. Typically, this is seen as a Sequence flow between two activities, without a conditional indicator (mini-diamond) or any intervening Gateway.