# UML™ Profile and Interchange Models for Enterprise Application Integration (EAI) Specification

**OMG Formal Specification**
**March 2004**
**formal/04-03-26**

OBJECT MANAGEMENT GROUP

**An Adopted Specification of the Object Management Group, Inc.**

Copyright © 2001, DSTC
Copyright © 2001, Hitachi, Ltd.
Copyright © 2001, IBM Corporation
Copyright © 2004, Object Management Group, Inc.
Copyright © 2001, Oracle Corporation
Copyright © 2001, Rational Corporation
Copyright © 2001, Unisys Corporation

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

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

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

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

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

COMPLIANCE

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

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

ISSUE REPORTING

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

# *Contents*

# Contents

# Contents

# Contents

# *Contents*

# Preface

## About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG’s objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

### What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

# OMG Documents

The OMG documentation is organized as follows.

## OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.
- ***Common Warehouse Metamodel (CWM) Specification*** mainly consists of definitions of metamodels in the following domains:
  - Object model (a subset of UML)
  - CWM foundation
  - Relational data resources
  - Record data resources
  - Multidimensional data resources
  - XML data resources
  - Data transformations
  - OLAP (On-line Analytical Processing)
  - Data mining
  - Information visualization
  - Business nomenclature
  - Warehouse process
  - Warehouse operation

## Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

## CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

### *OMG Interface Definition Language (IDL) Mapping Specifications*

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

### *CORBAservices*

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction*.

### *CORBAfacilities*

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility*.

## *Object Frameworks and Domain Interfaces*

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.

- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Life Science*: Comprised of specifications that relate to the OMG-compliant interfaces for the life science industry.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

## Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

# Typographical Conventions

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

**Helvetica bold** - OMG Interface Definition Language (OMG IDL) and syntax elements.

**`Courier bold`** - Programming language elements.

Helvetica - Exceptions

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

## *Acknowledgments*


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

- CBOP
- Charles Schwab & Co.
- Data Access Technologies
- DSTC
- Hitachi, Ltd.
- International Business Machines Corporation
- IONA
- Oracle Corporation
- Rational Software Corporation
- Unisys Corporation

# *Part 1 - Introduction*

## *Contents*

This section contains the following chapters.

# Introduction and Guide 1

## 1.1 Introduction

As enterprises adapt to business change and new opportunities, they seek to build on their existing strengths and assets for competitive advantage. Electronic trading with consumers and other businesses is one of these trends. This frequently entails building new applications by coupling existing ones, which is known as Enterprise Application Integration (EAI). This is most often done with some form of messaging that provides loose coupling to make it easy to change, to link heterogeneous systems and operating environments, and to maximize resilience and robustness in cases of partial failure.

Enterprise Application Integration technology is being promoted to integrate legacy systems with new packages. But integrating legacy applications with new software is a difficult and expensive task due, in large part, to the necessity of customizing each connection that ties together two disparate applications. There is no single mechanism to describe how one application may allow itself to be invoked by another.

We intend to solve this problem by defining and publishing a metadata interchange standard for information about accessing application interfaces. The goal is to simplify application integration by standardizing application metadata for invoking and translating application information. Once these standards exist, tools may be constructed to facilitate the development, execution, and management of these integration points.

Such connected systems are inherently complex to define and manage. A well-known approach to managing complexity is to define levels of concern. Modeling with UML has been shown to be successful at representing differing levels of detail. The appropriate level for EAI is *application architecture* — the treatment of the interfaces and interactions between applications. UML has been used successfully for modeling at this level, and this specification presents the authors' view of best practice for using the existing UML for modeling application architectures, i.e., architectures composed by enterprises to enable application integration.

## 1.2 Attachments

XMI and DTD files for the EAI Metamodels can be found in OMG document number ad/2001-08-25.

# Scope 2

The scope is described with three generic scenarios representing the evolution of the integration requirements:

- Scenario 1. Application integration through connectivity.
- Scenario 2. Application integration through information sharing.
- Scenario 3. Application integration through process collaboration.

For each scenario major characteristics and requirements are described. Obviously, scenario 2 requires the functionality described in scenario 1 and scenario 3 requires the functionality described in scenario 2. However, as we move forward from scenario 1 to scenario 2 and scenario 3, the underlying functionality becomes less visible and more and more hidden in the infrastructure.

As the industry moves forward, scenario 3-or an updated version of scenario 3-will most likely become the dominant scenario.

## 2.1 Scenario 1: Connectivity

A small set of applications has to communicate synchronously or asynchronously with each other to provide business functions.

It must be possible to model the following abstractions:

- Service requester and provider.
- Synchronous and asynchronous service request.
- Request, reply, and notification.

In this scenario, the participating applications share a common architecture. They share the data model of the communication and they are able to activate the appropriate applications to obtain a service.

There is a need for additional abstractions such as queues (local or remote) and topics. At one level, queues and topics should be invisible, but at a lower level of detail they may well be required.

## 2.2 Scenario 2: Information Sharing

This scenario comes from handling securities.

An investor orders a stock trade, typically by sending a message describing the stock trade to be carried out. (We discuss the creation of the message in the next scenario.) This stock trade order triggers a set of autonomous actions: checking the investor's account, checking the position of the institution, notifying a broker if the trade is large, and notifying a broker as well as the investor if there are any issues.

If the order is accepted, the market place is selected and an institution such as a market maker executes the trade. After execution, the investor records are updated. Information about the executed trade is sent to the investor via pager and e-mail and to internal systems such as bookkeeping that require the information.

The securities firm is not only interested in handling requests properly but also in answering questions from investors, regulators, and other interested parties, both internal and external, at any stage during or after a trade.

A key requirement is that it should be easy to add new participants and new functionality with no or minimal impact to existing participants and services.

A good way to deal with this scenario is to model it as information sharing between applications and actors, such as investors and brokers. Such information sharing can be implemented through publishing and subscribing to business events enabling communication between the participants. We assume that all applications reflect a shared understanding about the meaning and sequence of the individual business events and act according to this shared understanding. However, we will assume that applications and actors participating in these processes are isolated from knowledge about who will consume their information and in which topic and format the recipients expect it.

It must be possible to model the following abstractions:

- Messages representing business events. (We are much less interested in messages that do not represent business events.)
- Publication of messages and business events - the ability to share information.
- Queues and topics - it must be possible to separate output containers of sending applications from input containers of receiving applications.
- Data transformation - each program must be able to create or consume messages in its own format. Applications should be able to use data structures suitable to their own language, e.g., a C++ program should not have to handle SWIFT or XML formats. Data transformation has to include data verification.
- Propagation - the ability to use any protocol to receive or deliver a message, including the allocation of a received message to a queue.

- Subscriptions to determine the receiving programs, their input containers or propagation routes, and their transformations. Subscriptions should be able to represent various cases, including interest of users, data routing, activation of programs.
- Retention to keep the history of relevant messages from creation through stages of processing, transformation, and consumption.
- Auditing, tracking, and mining - the ability to find and relate messages, both consumed and in flight.

In this scenario, the applications share a common business event and process model at the conceptual level. However, details of the layout of the data may vary, e.g., one program may use SWIFT structures, while another uses XML.

The term information sharing is used to characterize the interaction between participants providing information for the right recipients. Where time is of the essence and information is communicated with messaging/event technology we refer to zero latency information sharing.

## 2.3 Scenario 3: Process Collaboration

Company A offers its merchandise through the Internet. While some customers order goods using a browser interface, the majority of the orders are communicated business-to-business (B2B) using one of the B2B protocols. In simplified form, a B2B protocol consists of the following business events:

- RFQ (Request For Quotation)
- Offer
- Acceptance
- Shipment notice
- Bill
- Payment

Other events involved in negotiations, inquiries, changes, cancellation, and other additional steps (e.g., steps involving communications problems) are not considered in this simplification.

Company A communicates with business partners over secure Internet channels. Non repudiation, high reliability (including disaster tolerance), exactly once semantics, fully automated user-accessible application-independent auditing and tracking are basic requirements. Outgoing communication will use the requested protocol. Messages representing business events are carefully checked for process, sequence and data accuracy. Any error will raise an exception condition. Incoming communication is checked carefully as well. Some errors may need manual correction, which needs careful documentation.

Company A offers the flexibility for customers to use their favorite B2B protocols as long as they can represent a proper order process.

Applications should be independent of the specifics of the business protocols, but it is assumed that the desired interaction with an application can be achieved using its interfaces. At least three levels of interface support can be distinguished in applications:

- Applications that are only able to react through activation of their interfaces.
- Applications that can accept requests and can notify the outside world using events. At least some of these applications have to be configured to activate the desired events.
- Applications that additionally provide a process interface. These applications have to be configured to use the desired process structure.

In any of these cases it cannot be assumed that the process as seen by the application is the process as seen by the selected B2B protocol. Actually it is desirable to hide the internal processes from business partners, so they can be changed without impact to the outside world and potential competitive advantages can be hidden. To achieve this, a mediation service has to be available to transform the process and data semantics embedded in the B2B protocols to the process and data semantics of an enterprise's internal processes. This transformation will be called semantic mediation. Semantic mediation is part of the core functionality required for the integration of autonomous applications.

Flexibility in B2B communication requires a repository of information that governs communication with a particular trading partners. This information includes security (including application security), notifications, subscriptions, B2B protocols and their extensions and adaptations, and indications for internal routing. It should be possible to group trading partners according to various criteria. This information comprises what is often termed a trading community agreement.

To model this level of integration it must be possible to model the following abstractions in addition to the abstractions defined in the previous scenarios:

- Semantic mediation - the ability to transform process and data structures between applications and B2B protocols.
- Propagation between enterprises - secure, with non repudiation, exactly once semantics, and disaster protection.
- B2B-level auditing, tracking, and mining-a business event can be reviewed, analyzed in its process context, and mined for insight into business behavior.
- B2B protocols - processes based on the communication of business events or business events in the context of process and customer relations.
- Trading community agreements in a repository containing information about trading partner and the communications with them.

This specification addresses primarily the first two scenarios. It provides enablement for scenario 3, but this scenario requires other elements that go beyond its scope. Scenario 3 is included here to clarify the relationship to work going on in ebXML and elsewhere.

# Modeling Approach 3

The EAI specification is delivered as a complete MOF-based metamodel and a UML profile, which actually consists of two profiles, one for collaboration modeling and one for activity modeling. This approach facilitates exchange with both UML tools and MOF-based tools/repositories.

## 3.1 Metamodel

As is the common practice, the MOF-based metamodel is captured as an object-oriented model expressed using a suitably restricted subset of the UML notation. The UML elements used in this specification are:

- Classes with attributes and (query) operations.
- Binary associations, where composite and navigation adornments are permitted. Association classes and qualified associations are not permitted.
- Packages, including nesting and imports.
- The object constraint language, OCL, for expressing well-formedness constraints.

The EAI metamodel is documented using the following conventions:

- The overall structure of the metamodel is shown as one or more package diagrams, depending on the level of nesting required.
- At the lowest level, packages are limited in size, and only one class diagram per package is required.
- In explaining a package, the important collaborations between classes are identified and described as one. Individual classes are described separately where this enhances the overall understanding of the model.
- Well-formedness constraints are also grouped with the collaborations to which they are relevant.
- The semantics of each collaboration is described as specified below.

## 3.2 UML Profile

The UML profile allows modelers to use UML as a concrete notation for producing EAI models using UML modeling tools that support the UML extensions mechanisms, chiefly stereotypes, tagged values, and custom icons. Some tools are available, e.g., [ref objecteering], which can accept a profile definition and configure a modeling tool to force modelers to conform to that profile by using only elements of the UML subset and only the stereotypes, tagged values, and icons declared in the profile.

A mapping between the metamodel and the UML profile is defined as part of the EAI specification. This is intended as a basis for the development of tools that will transform models expressed using the UML profile into models conforming to the metamodel, and vice versa. The details of the mapping are given as part of the definition of the profile.

## 3.3 Four-layered Architecture

The relationship of the EAI specification to the four-layered architecture defined by the OMG is as follows. MOF is at level 3, so the EAI metamodel is at level 2. The EAI UML profile is also at this level - it is just a set of additional constraints (what stereotypes, tagged values, etc.) on how UML is to be used when notating EAI models. The EAI metamodel should be thought of as the definition of the abstract syntax of EAI models. An EAI model, which is at level 1, is an expression of this abstract syntax. An EAI model is a specification of the architecture of an event-based system and the allowable information flows through that system. Level 0, then, represents actual behaviors of an event-based system, for example a particular instantiation of the architecture or a particular message flow through that system. These behaviors and instantiations must conform to the specification of behavior captured by the EAI model.

## 3.4 Semantics

There are a number of approaches to semantics. One is to describe how a model (in this case an EAI model) constrains the set of possible behaviors at M0 which satisfy that model. This can be captured formally by explicitly modeling (in some formal language) the structure of the abstract syntax, the structure of M0 behaviors and the relationship between the two. However, a formal definition can be somewhat inaccessible. The approach taken in this specification is to describe the semantics in English, using a model of M0 behaviors to help clarify the explanation where appropriate.

# Compliance 4

## 4.1 Overview

Compliance with this standard by a vendor can be partial. To facilitate this the compliance points have been defined separately (Section 4.2, “Compliance with the UML Collaboration Profile,” on page 4-1, Section 4.3, “Compliance with the UML Activity Profile,” on page 4-2, and Section 4.4, “Compliance with the MOF-based EAI Metamodel,” on page 4-2) and examples of plausible compliance statements are provided (Section 4.5, “Compliance Statement Examples,” on page 4-3).

References to other OMG standards are abbreviated in the compliance point definitions, but in all cases refer to the specific revisions listed in the table below:

| **Standard** | **Version Referenced** |
|---|---|
| UML | 1.4 |
| XMI | 1.2 |
| MOF | 1.3 |

## 4.2 Compliance with the UML Collaboration Profile

The UML Collaboration Profile for EAI is defined in Chapter 8.

### 4.2.1 General Compliance

A compliant implementation supports the UML XMI exchange mechanism for the UML packages extended by the Collaboration Profile for EAI. It also supports the UML exchange mechanism for the stereotypes and tagged values defined by the Profile.

The UML packages that the Profile extends are "Behavioral Elements::Collaborations" plus the transitive closure of all of the packages upon which that package depends.

An implementation that satisfies the General Compliance point can be described as one that "complies with the UML Collaboration Profile for EAI."

### 4.2.2 Visualization

An implementation satisfies the visualization compliance points if it supports the UML notation for the packages extended by the Collaboration Profile and for the EAI extensions to those packages. An implementation that complies with the Collaboration Profile may or may not satisfy the Visualization compliance point.

An implementation that complies with the Collaboration Profile and that satisfies the Visualization compliance point for the Profile can be described as one that "complies with the UML Collaboration Profile for EAI including UML notation."

## 4.3 Compliance with the UML Activity Profile

The UML Activity Profile for EAI is defined in the Activity Modeling chapter.

### 4.3.1 General Compliance

A compliant implementation supports the UML XMI exchange mechanism for the UML packages extended by the Activity Profile for EAI. It also supports the UML XMI exchange mechanism for the stereotypes and tagged values defined by the Profile.

The UML packages that the Profile extends are "Behavioral Elements::Activity Graphs" plus the transitive closure of all of the packages upon which that package depends.

An implementation that satisfies the General Compliance point can be described as one that "complies with the UML Activity Profile for EAI."

### 4.3.2 Visualization

An implementation satisfies the visualization compliance points if it supports the UML notation for the packages extended by the Activity Profile and for the EAI extensions to those packages. An implementation that complies with Activity Profile may or may not satisfy the Visualization compliance point.

An implementation that complies with the Activity Profile and that satisfies the Visualization compliance point for the Profile can be described as one that "complies with the UML Activity Profile for EAI including UML notation."

## 4.4 Compliance with the MOF-based EAI Metamodel

There is a separate and independent compliance point for each of the MOF metamodels defined in this specification.

A compliant implementation of a metamodel supports exchange based on the XMI DTD generated from the metamodel.

The metamodels defined by the specification and the corresponding generated XMI DTDs are as follows:

| **EAI MOF based metamodel** | **Chapter in which the metamodel is defined** | **XMI DTD** |
|---|---|---|
| Integration | 6 | CM4EAI |
| TDLang | 7 | TDLang |
| TypeDescriptor | 7 | TypeDescriptorTDLang |
| COBOL | 14 | COBOLtdlang |
| PL/I | 14 | pliTDLang |
| C | 14 | tdlang |
| C++ | 14 | cpptdlang |

The language metamodels depend on the TDLang and typedescriptorTDLang XMI DTDs.

There are no specific requirements for visualization of the EAI Metamodel.

A compliant implementation of the Integration metamodel can be described as one that “complies with the EAI Integration metamodel;” a compliant implementation of the COBOL metamodel can be described as one that “complies with the EAI COBOL metamodel;” etc.

## *4.5 Compliance Statement Examples*

Any combination of the compliance points can be used. Examples of compliance statements follow:

- Tool XXX complies with the UML Collaboration Profile for EAI.
- Tool XXX complies with the UML Collaboration Profile for EAI including UML notation.
- Tool XXX complies with the UML Activity Profile for EAI.
- Tool XXX complies with the UML Activity Profile for EAI including UML notation.
- Tool XXX complies with the UML Collaboration and Activity Profiles.
- Tool XXX complies with the UML Activity Profile including UML notation and with the UML Collaboration Profile.
- Tool XXX complies with the UML Collaboration Profile including notation and with the UML Activity Profile including notation.
- Tool XXX complies with the UML Collaboration and Activity Profiles, including UML notation for both. (Note: this statement is equivalent to the previous one.)

- Tool XXX complies with the EAI C Metamodel.
- Tool XXX complies with the EAI C++ Metamodel .
- Tool XXX complies with the EAI Integration, C, C++, and PL/I metamodels.
- Tool XXX complies with the UML Collaboration and Activity Profiles including notation for both. It also complies with the EAI Integration, COBOL, and PL/I metamodels.

# Relationships to Other Standards 5

## 5.1 Relationship to Envisioned OMG Technology

This section describes the relationship, in terms of alignment, reuse or overlap with OMG standards for which RFPs have been issued but which have not yet been adopted.

### 5.1.1 Real-time

The UML Profile for Scheduling, Performance and Time (from the Real-time PSIG, OMG document number ad/01-06-14) emphasizes the definition of quality of service (QoS). The UML Profile for EAI makes provision for QoS specifications in the provision of streams (Section 6.4.1.2, "EAIStream," on page 6-23) and resources (Section 6.3.7, "EAIQueue," on page 6-9). These are left non-specific in this specification and can be augmented with specifications from the UML Profile for Scheduling, Performance, and Time.

## 5.2 Relationship to Existing Standards

### 5.2.1 UML

As a UML profile, this specification defines uses of UML 1.4 for the purposes of application integration. This includes classes and stereotypes.

### 5.2.2 Meta Object Facility (MOF)

UML is MOF compliant. This specification defines UML elements and adds additional semantics appropriate to the context of event-based architectures in EAI. Chapter 2 presents a metamodel in which each class is a MOF Class instance at the M2 level.

### 5.2.3 Common Warehouse Metamodel (CWM)

The Common Warehouse Metamodel (CWM) defines and publishes a metadata interchange standard for data warehousing and business intelligence tools and resources.

CWM gives metamodels for generic data structures that include XML documents, COBOL records, C structures and SQL schemas. These are aimed at data stores but are generic. They could be applied to message content descriptions. This level of refinement is a natural progression from the architectural designs supported by the UML Profile for Event-based Architectures in EAI.

CWM is highly reusable and is independent of any particular tool or data resource. It reduces the work required to integrate data warehousing and business intelligence tools.

CWM is needed for data transformation in a data warehousing and business intelligence environment. It provides data type mapping between a mix of different data resources, facilitates data translations from one data resource into another, allows data driven impact analysis for data lineage and allows data resource schemas to be viewed by developers.

The EAI Common Application Metamodel (CAM), which is described in Chapter 7, defines and publishes a metadata interchange standard for information about accessing enterprise applications such as CICS and IMS. CAM is reusable and is independent of any particular tool or middleware. It is likely to provide an incentive to connector suppliers by reducing the work required to create and develop connectors and/or connector builder tools.

CAM is needed for data transformation in an enterprise application integration environment. It provides data type mapping between mixed languages and facilitates data translations from one language and platform domain into another, it will allow data driven impact analysis for application productivity and quality assurance, and it will allow programming language data declarations to be viewed by developers.

In CAM a language metamodel, such as the COBOL metamodel, is used by enterprise application programs to define data structures which represent connector interfaces. It is important for connector tools to show a connector developer the source language, the target language and the mapping between the two. The CAM language metamodel also includes the declaration text in the model. This permits the connector/adapter developer to see the entire COBOL data declaration, including comments and any other documentation that would help him/her understand the business role played by each field in the declaration.

While CWM focus on data resources, CAM is for applications. CWM and CAM complement each other; both are needed in an enterprise IT environment.

## 5.3 Other Related Activities

Specification may deal with business-to-business (B-to-B) models as well as intra-enterprise models. However, there are other significant standards activities in B-to-B, and this specification does not address the area directly. EAI is a valuable underpinning to B-to-B along with other facets such as process modeling, which is addressed to a certain extent in the UML Profile for EDOC Business Processes Profile. To offer public services and interfaces to trading partners, an enterprise has to ensure that it has well-defined interfaces and well-architected systems. Much trading is inherently event based, and so streams, messages, publications, sources, targets, filters, transformations and other operations are natural modeling elements for the intra-enterprise systems that are needed to support both internal and public electronic trading. Hence, EAI is important both to inter and intra-enterprise business processes.

B-to-B modeling is dealt with in ebXML, which is based on a particular approach to B-to-B implementation. However, there are other approaches, including web services (SOAP, WSDL, UDDI and BPEL4WS) at W3C and OASIS, RosettaNet, OBI, EDI, OAG BODs and several industry-specific formats and protocols. BPML is a rival to BPEL4WS, which can be used to specify workflow and intra-enterprise processes as well as inter-enterprise processes. There continues to be a high volume of activity and a rapid rate of change.

# *Part 2 - Metamodel*

## *Contents*

This section contains the following chapters.



This section describes the EAI metamodel, which, as explained in Chapter 6, is MOF compliant. The metamodel captures the essential EAI concepts. It may also be viewed as the abstract syntax of a language for specifying architectures for enterprise application integration. The metamodel is in two sections:

- The Integration Metamodel dealing with connectivity, composition, and behavior.
- The Common Application Metamodel dealing with interfaces and formats.

Chapter 7 describes a UML profile for the language of the Integration Metamodel. It defines how UML (and therefore UML modeling tools) can be used as a concrete notation for this language.

# EAI Integration Metamodel 6

## 6.1 EAI Integration Specializes FCM

The EAI Integration metamodel is a specialization of the Flow Composition Model from the UML Profile for EDOC (OMG Document Number: formal/04-02-03). The following sections make extensive use of terms described in the FCM, and consequently it is assumed that the reader is familiar with it.

The UML Profile for EDOC also presents the Component Collaboration Architecture (CCA), part of the EDOC Enterprise Collaboration Architecture (formal/04-02-01). In Section 6.5, "CCA Component Library for EAI," on page 6-38 a mapping is presented between EAI Integration metamodel and the CCA. The mapping introduces the concept of a CCA "Component Library." Many of the concepts in EAI are represented as standard components that may be used in EAI compositions.

The EAI Integration metamodel reuses the concepts of flow, flow node, and composition. It adds the following basic concepts that are required in EAI architectural modeling:

- Asynchronous communication
- Message queuing
- Message content and format

It additionally uses the FCM to define as flow components a number of concepts common to the message oriented middleware used in EAI, such a message routing, transformation, and publish/subscribe communication.

## 6.2 FCM support for recursive composition

The UML profile for EDOC provides support for the definition of 'composite nodes,' whose function is defined by an FCMComposition. The FCMNodes in an FCMComposition may themselves be composite. Terminals on a composite FCMNode have an association with either an FCMSource or an FCMSink in the FCMComposition that defined an FCMCompositeNode. This is detailed in the ‘UML Profile for EDOC/Flow Composition Model’ document number formal/04-02-03.

## 6.3 EAI Specializations of the FCM

### 6.3.1 Motivation

This section defines a set of specializations of the FCM. Each of these introduces a new concept required for EAI architectural modeling.

### 6.3.2 EAILink

***Definition***

Links between entities in an EAI architecture are often treated as event channels, and the occurrence of an event on such a channel initiates processing of the information associated with the event. As such, these links represent the flow of both data and control. In the FCM, data and control links are separate, so we introduce EAILink. EAILink inherits from FCMDataLink (which is a terminal to terminal link), and has an association with a single FCMControlLink.

Links may have their synchronization specified as synchronous, in which case a link between a pair of terminals implies a synchronous (call) invocation of the relevant FCMOperation, or asynchronous in which case a link between a pair of terminals implies an asynchronous invocation of the relevant FCMOperation (the FCMOperation that owns the parameter that the terminal represents).

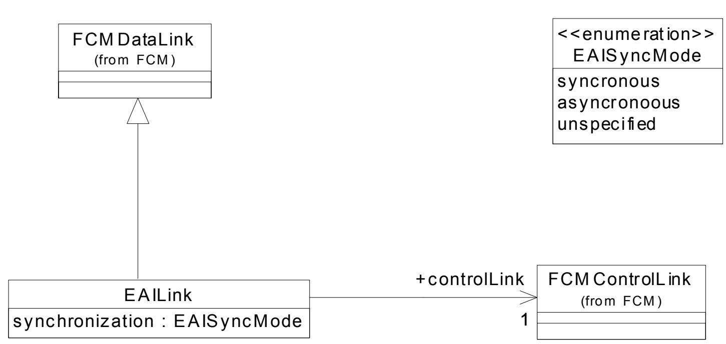



*Figure 6-1* Definition of EAILink

### *Constraints*

The source terminal of the EAILink is the same as the source terminal of its *controlLink*:

**context** EAILink **inv**:

self.sourceTerminal = self.controlLink.targetTerminal

The target terminal of the EAILink is part of the *interface* of the *targetNode* of the controlLink:

**context** EALink **inv**:

self.controlLink.targetNode.interface->exists(t | t=self.targetTerminal)

An EAILink connects two EAITerminals:

**context** EAILink **inv:**

**inv:** self.sourceTerminal.oclIsKindOf(EAITerminal)

**inv:** self.targetTerminal.oclIsKindOf(EAITerminal)

An EAILink connects two EAI Operators, sources or sinks:

context EAILink

inv: self.sourceNode.oclIsKindOf(EAIOperator) **or**

self.sourceNode.oclIsKindOf(EAISource)

inv: self.targetNode.oclIsKindOf(EAIOperator) **or**

self.targetNode.oclIsKindOf(EAISink)

## *6.3.3 EAITerminal*

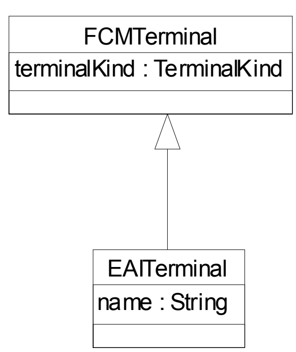



*Figure 6-2* EAITerminal

#### Definition

An EAITerminal is a specialization of FCMTerminal.

#### Constraints

EAITerminal can be connected to other instances of terminals only via instances of EAILink.

(Any link that can have a source terminal that is an EAITerminal must be an EAILink, any link that can have a target terminal that is an EAITerminal must be an EAILink)

context FCMComposition

inv: self.connections->forall(c | if c.oclIsTypeOf(FCMTerminalToNodeLink) then c.sourceTerminal.oclIsKindOf(EAITerminal) implies c.oclIsKindOf(EAILink))

inv:self.connections->forall(c | if c.oclIsTypeOf(FCMTerminalToTerminalLink) then c.targetTerminal.oclIsKindOf(EAITerminal)

An EAITerminal is the representation (see Figure 6-2) of an FCMParameter that is of type EAIMessageContent.

### 6.3.4 EAIMessageParameter

#### Description

An EAIMessageParameter defines the data to be processed by an EAIOperation. It is used to model an EAI message.

An EAIMessageParameter conforms to a message format specification, which may be physically manifest in the message (as, for example, with an inline XML DTD) or may need to be inferred by the MOM infrastructure. In order to make this kind of distinction, EAIMessageContent has two properties:

- **domain** - specifies the most generic message wireformat domain, and could be considered to encompass the domain of a generic parser. This is not restricted, but examples such as ‘XML,’ ‘FixedFormat,’ ‘Delimited’ would be valid.
- **name** within the domain specified above - this is the name of the message format to be processed. This information is intended to allow message format handling infrastructure to identify what type of message within a particular domain is being processed, for example by reference to an XML Schema Document.

Subclasses of EAIMessageParameter can optionally be used to specify further structure in a message.

EAIMessageParameter inherits an association with TDLangElement, which may be used to specify details of the physical rendering of the message.

1. The TDLang metamodel provides an abstract view of the message element’s structure. It may be used to represent both primitive and more complex data structures.

2. TDLang provides access to the language-specific representation of the message element (via the COBOL, PL/I, and other language metamodels in CAM), as well as its physical wire format (via the Type Descriptor Metamodel in CAM).

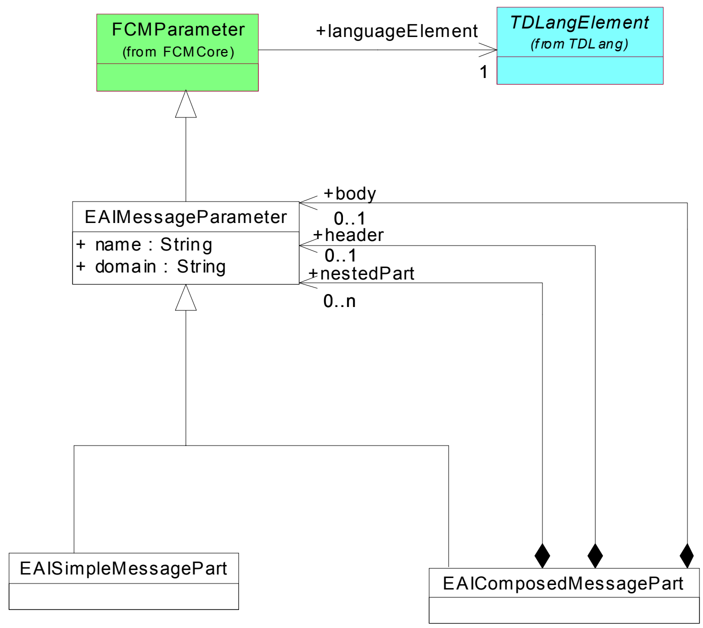



*Figure 6-3* EAI MessageContent

### 6.3.4.1 *EAISimpleMessagePart*

Models a message part that does not contain other message parts. The format of a message element is defined in the MessageContent metamodel by its association to a TDLangElement, which is a class in the CAM (see Chapter 7). This link into the CAM provides all of the following for message elements.

### 6.3.4.2 *EAIComposedMessagePart*

Models a message or part of a message that is further composed of subparts. The model may optionally specify sub-parts that are considered to be ‘header’ or ‘body’ information.

### 6.3.4.3 *EAIHeader*

It is a common requirement for message processing to be able to specify a location to send any potential replies to, and to specify a location to which to send a message in the event of a message processing error. The information required to do this can be specified via a subclass of EAIHeader. In cases where the metadata contained in a header element does not concern replies or exceptions, it is not required for all headers in EAIMessageContent to be subclasses of EAIHeader.

EAIHeader is a subclass of EAIMessageElement; it has two associations to EAIMessageElement:

- replyTo: an EAIMessageElement that is required to specify the terminal to which replies to an instance of a message should be sent.
- exceptionTarget: an EAIMessageElement that is required to specify the terminal to which exception notices should be sent.

### *Constraint*

The exceptionTarget and replyTo EAIMessageElement must not themselves be instances of the subclass EAIHeader.

**context** EAIHeader

**inv:** replyTo->forall(rto | rto.oclIsKindOf(EAIHeader) = false)

**inv:** exceptionTarget->forall(exc | exc.oclIsKindOf(EAIHeader) = false)

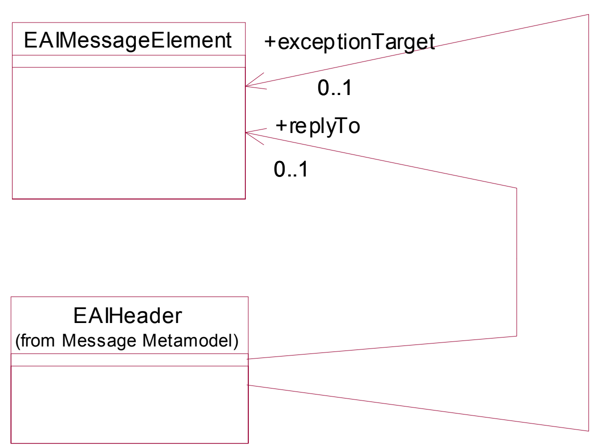



*Figure 6-4* EAIHeader

## 6.3.4.4 *EAIExceptionNotice*

Messages of this form may be sent if an exception occurs during the processing of a message. An instance of an ExceptionNotice will normally contain the original message, with additional exception-specific information in a separate message part. In addition to these required message parts, the message may contain other message parts. These may be specified using the association to MessagePart inherited from EAIMessageContent.

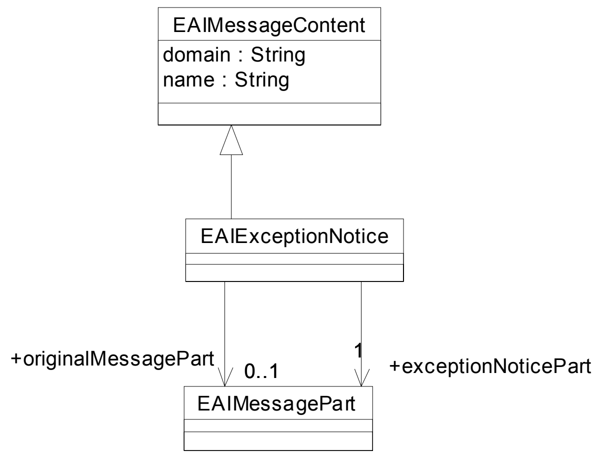



*Figure 6-5* EAIExceptionNotice

#### *6.3.4.5 EAIMessageContent*

#### *6.3.4.6 EAIMessagePart*

EAIMessagePart may have two distinct elements:

1. A message header that contains metadata about the message rather than the application data itself. It is used to help determine processing either by middleware or by metadata-aware applications.

2. Message body, which contains the business content of the message.

The header and the body modeled via associations with EAIMessageElements.

#### *6.3.4.7 EAIComposedMessagePart*

EAIComposedMessagePart is a subclass of EAIMessagePart, which may itself contain messageparts.

### *6.3.5 EAIMessageOperation*

***Description***

EAIMessageOperation is a subclass of FCMOperation used to describe operations for which all the inputs and outputs are messages.

### Constraints

Every input and output of an EAIMessageOperation is an EAIParameter. EAIMessageOperation may have zero or one faults. If present, the fault must be an EAIParameter.

**context** EAIMessageOperation

inv: self.inputs->union(self.outputs)->forAll(oclIsType(EAIParameter))

inv: self.inputs->union(self.faults)->forAll(oclIsType(EAIParameter))

**inv:** self.inputs->size() <= 1

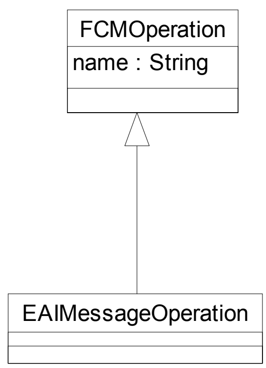



*Figure 6-6* MessageOperation

## 6.3.6 EAISource and EAISink

### Description

EAISource and EAISink represent points in an EAI architecture where messages appear (EAISource) and disappear (EAISink).

Sources and sinks may make use of EAIResources. An EAIResource represents a usable and sharable entity such as a queue (Section 6.3.7, “EAIQueue,” on page 6-9 or a database (Section 6.4.1.5, “EAIDBTransformer,” on page 6-26).

### Constraints

EAISource is a subclass of FCMSource. Its sinks must be EAISink, and its implements operation must be an FCMOperation.

EAISink is a subclass of FCMSink. Its source must be an EAISource.

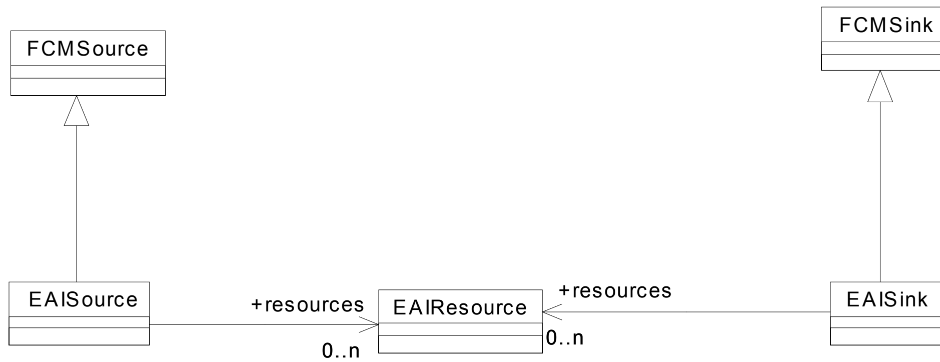



*Figure 6-7* Sources and sinks

## 6.3.7 EAIQueue

***Description***

EAIQueue is a queue of finite or unbounded length, and is modeled as a subclass of EAIResource.

EAIQueue has a name and a Boolean "isBound" showing if the queue length is finite or unbounded. EAIQueue also has a maxLength, which specifies the maximum number of messages it can hold.

EAIQueue is restricted to holding a specific type of message content if an EAIMessageContent is specified for EAIQueue. Otherwise, EAIQueue can hold any type of message content.

EAIQueue is intended to be an abstraction of queuing infrastructure. We note that most MOM implementations allow machine-to-machine communication via a remote queuing infrastructure that can specify a number of different queue types and relationships between them. This can be modeled as refinement or realization of EAIQueue (see Section 6.4.1.2, "EAIStream," on page 6-23) or of the EAIPrimitiveOperator EAIStream.

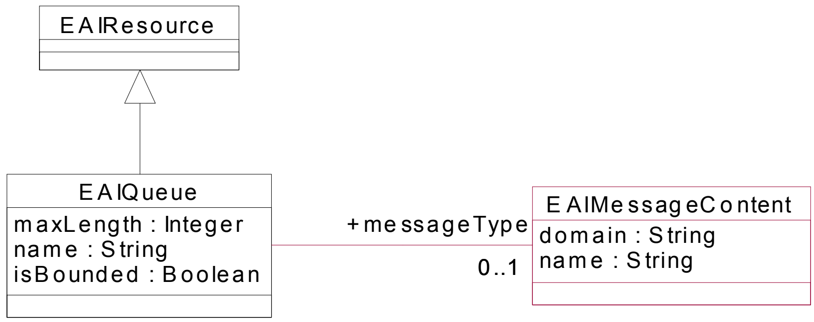



*Figure 6-8* EAIQueue

***Constraints***

maxLength >= messages->size()

## *6.3.8 EAIQueuedInputTerminal and EAIQueuedOutputTerminal*

A common means of implementing an asynchronous link between a pair of entities in EAI is for them to share a queuing infrastructure. In this case, the entity in which an event occurs places a message into a queue and then continues processing. The entity that is to act on this information can remove the message from the queue at any time. This normally involves the receiving entity doing one of the following:

1. Polling the queue for the arrival of a message.
2. Blocking execution awaiting the arrival of a message.
3. Being triggered by the arrival of a message.

EAIQueuedInputTerminal and EAIQueuedOutputTerminal are subclasses of EAITerminal that are used to represent message communication that occurs via queuing.

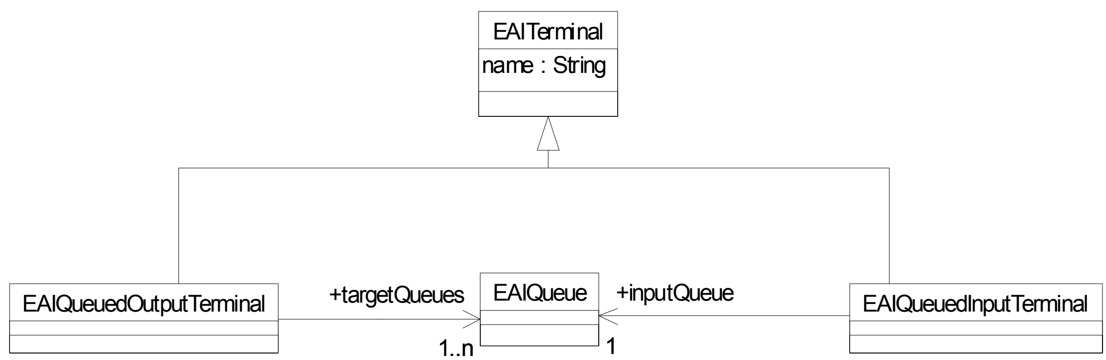



*Figure 6-9* EAIQueuedOutputTerminal and EAIQueuedInputTerminal

An EAIQueuedInputTerminal has an association with the single queue that it reads from, while an EAIQueuedOutputTerminal has an association with each of the queues used by its target EAIQueuedInputTerminals.

Any operator that has an EAIQueuedOutputTerminal is understood to place a single copy of its output message on each of its targetQueues.

Queued input and output terminals may be used on any of the EAI constructs that have terminals (EAIPrimitiveOperator, EAICompoundOperator, EAISource, EAISink).

### *Constraints*

All EAILinks from an EAIQueuedOutputTerminal as the sourceTerminal must have an EAIQueuedInputTerminal as the TargetTerminal.

The EAILink from an EAIQueuedOutputTerminal to an EAIQueuedInputTerminal must have synchronization=asynchronous.

An EAILink between an EAIQueuedOutputTerminal and an EAIQueuedInputTerminal implies that the inputQueue of the inputTerminal is in the targetQueues of the output terminal.

All EAIQueuedInputTerminals have EAILinks with all EAIQueuedOutputTerminals that use the same queue instance.

**context** EAILink

**inv:** if self.sourceTerminal.oclIsKindOf(EAIQueuedOutputTerminal) **then**

self.targetTerminal.oclIsKindOf(EAIQueuedInputTerminal) **and**

self.synchronization=asynchronous **and**

self.sourceTerminal.targetQueues->includes(self.targetTerminal.inputQueue)

### 6.3.9 EAIQueuedSource and EAIQueuedSink

#### Description

EAIQueuedSource and EAIQueuedSink are used to model the internal elements of an EAIMessageFlow that is associated with EAIQueuedInputTerminals and EAIQueuedOutputTerminals.

When viewing the internals (i.e., the EAIMessageFlow) of a CompoundOperator, the element of the flow that receives messages (and passes them on to the rest of the flow) is a source of messages to the rest of the EAIMessageFlow, and vice versa. Hence, the part that reads from a queue is modeled as an EAIQueuedSource and the part that writes to a queue as EAIQueuedSink.

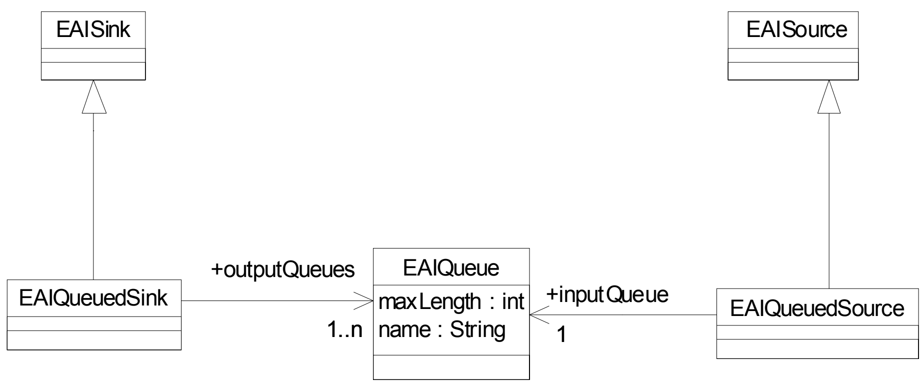



*Figure 6-10* QueuedSource and QueuedTarget

Note that the terminals of EAIQueuedSink and EAIQueuedSource (used within the EAIMessageFlow) could themselves be queued terminals. This would imply that queueing is used both outside and inside the EAIMessageFlow.

#### Constraints

The outputQueues of an EAIQueuedSink must be the same as the targetQueues of the EAIQueuedOutputTerminal that it is associated with.

The inputQueue of an EAIQueuedSource must be the same as the inputQueue of the EAIQueuedInputTerminal that it is associated with.

#### Refinement relationships

An EAILink with synchronization of unspecified is refined by an EAILink with synchronization of either synchronous or asynchronous.

Where there is an instance of an EAILink with a synchronization of asynchronous linking a pair of FCMTerminals, this is refined by the substitution of EAIQueuedInputTerminal and EAIQueuedOutputTerminal for the FCMTerminals.

## 6.3.10 Operators

### 6.3.10.1 EAIOperator

Operators act upon messages as they flow between systems. We define EAIOperator to be a subclass of FCMFunction.

EAIOperators have a type, EAIType. An EAIOperator prototype can also be used to specify an EAIType. EAIOperator may optionally specify EAIResources that it uses to enact its function.

#### *Constraints*

context EAIOperator

Define what it means to be a prototype

let isPrototype = self.defines->size() = 1

let isInstance = self.defines->isEmpty()

An EAIOperator has the same number of terminals as its prototype;

inv: if isInstance then self.interface->size() = self.type.protoype.interface->size()

The prototype for a prototype is itself;

inv: if isPrototype then self.type.prototype = self

All of the terminals of an EAIOperator are EAITerminals;

inv: self.interface->forall(t | t.oclIsKindOf(EAITerminal))

An EAIOperator's terminals have the same names as its prototype;

inv: if isInstance then self.interface->

forall(t | self.type.prototype.interface->exists(tt| tt.name=t.name))

An EAIOperator has the same set of resources as its prototype;

inv: if isInstance then self.resources = self.type.prototype.resources

### 6.3.10.2 EAIResource

EAIResource is used to model resources such as databases that are used by operators.

### 6.3.10.3 EAIType

EAIType is a subclass of FCMType. It may have a single EAIOperation. An EAIType is defined by a prototype EAIOperator.

#### Constraints

context EAIType

EAIType has single operation:

inv: self.operations->sizeOf() = 1

The single operation is the same as the ‘invokes’ operation of the prototype:

inv: self.operations->any() = self.prototype.invokes

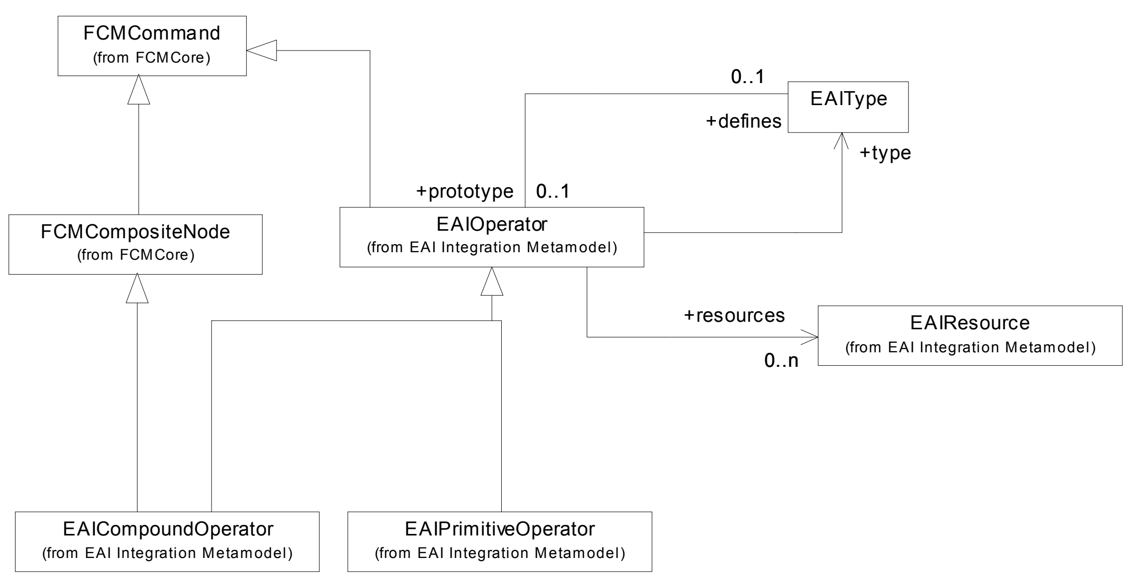



*Figure 6-11* Definitions of PrimitiveOperator and CompoundOperator

### 6.3.10.4 EAIPrimitiveOperator

#### Description

Instances of EAIPrimitiveOperator enact a simple message processing operation. EAIPrimitiveOperator is a subclass of EAIOperator.

#### Constraints

Inherited from EAIOperator.

### 6.3.10.5 EAICompoundOperator

#### *Description*

An instance of an EAICompoundOperator composes more complex message processing behavior from EAIPrimitiveOperators, from other EAICompoundOperators, or both. EAICompoundOperator inherits its 'composition' characteristics from FCMCompositeNode and its EAI-specific constraints from EAIOperator. Further constraints are described below.

#### *Constraints*

**context** EAICompoundOperator

The EAIType of an EAICompoundOperator must have an association with an FCMComposition:

self.type.fCMComposition->size() = 1

Define the implementingComposition derived association:

let implementingComposition = self.type.fCMComposition->any()

The implementingComposition must be an EAIMessageFlow:

implementingComposition.oclIsKindOf(EAIMessageFlow)

Define the *nodes* derived association:

self.nodes = self.implementingComposition.nodes

Define the FCMOperations implemented by the FCMComposition:

let sourceNodes =

self.implementingComposition.nodes->

select(n | n.oclIsKindOf(EAISource))

let sourceOperations = sourceNodes.implements

The operations implemented by the EAISource nodes in the composite are the same as the operations specified for the EAIType of the node.

**inv:** sourceOperations = self.type.operations

### 6.3.10.6 EAIMessageFlow

An EAIMessageFlow is a subclass of FCMComposition. Each of its nodes (see Figure 6-2 on page 6-3) must be one of the operator classes (EAIPrimitiveOperator or EAICompoundOperator), and its connections must be EAILinks. In addition it allows nodes to have explanatory annotations attached to them.

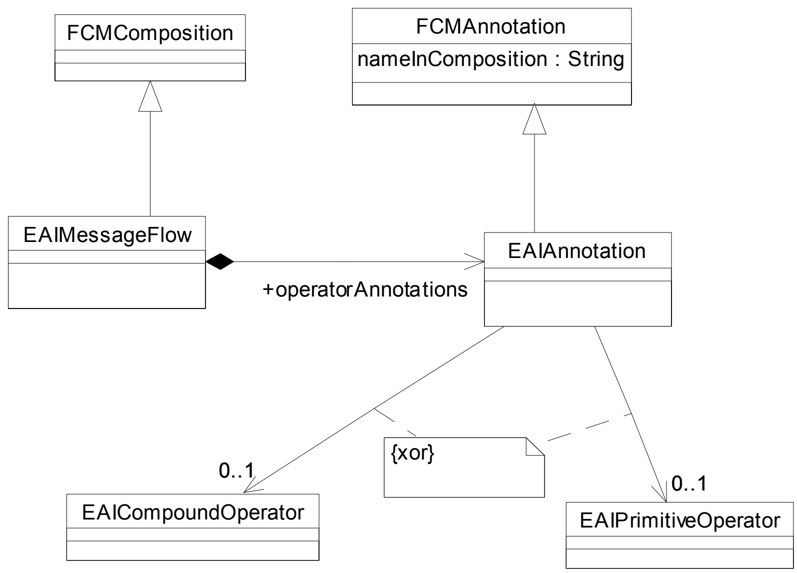



*Figure 6-12* EAIMessageFlow

#### *Constraints*

**context** EAIMessageFlow **inv:**

self.nodes->forall(n : n.oclIsOfKind(EAIPrimitiveOperator) or n.oclIsOfKinf(EAICompoundOperator))

## *6.3.11 Adapters*

An integration architecture provides paths for the flow of messages between the systems being integrated. Adapters provide the points at which the message-flow paths are actually connected to those systems. An adapter converts a specific kind of message from some system-specific format into a specified message-content type, or vice versa. EAI adapters are modeled as a specialization of FCMFunction.

### *6.3.11.1 EAISourceAdapter*

An EAISourceAdapter obtains information from a system, translates it into (some subclass of) EAIMessageContent and then sends it. Source adapters are modeled as a subclass of FCMFunction. The mapping between the internal format and the message is specified by an internalToMessage FCMMapping.

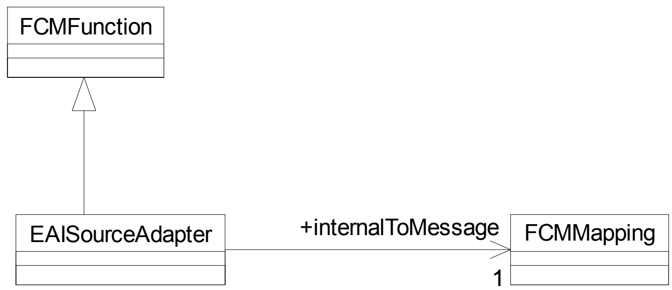



*Figure 6-13* SourceAdapter

#### *Constraints*

An EAISourceAdapter has a single output terminal, which is an EAITerminal with the name "out."

Output parameters of the invokes FCMOperation of SourceAdapter must be EAIParameters, which are associated with EAIMessageContent.

There is no constraint on the type of input terminals.

There is no constraint on the type of input and fault FCMParameters. It is noted that the faults FCMParameters may be EAIParameters (with EAIMessageContent) but that this is unlikely to be the case for input because adapters are used to link messaging to other (internal) interfaces.

### *6.3.11.2 EAITargetAdapter*

An EAITargetAdapter has a single input EAITerminal ("in"). It receives a message with content of a given input type, maps the message content to the format required for a system and then delivers the information to the system. The transformation is specified by a messageToInternal FCMMapping.

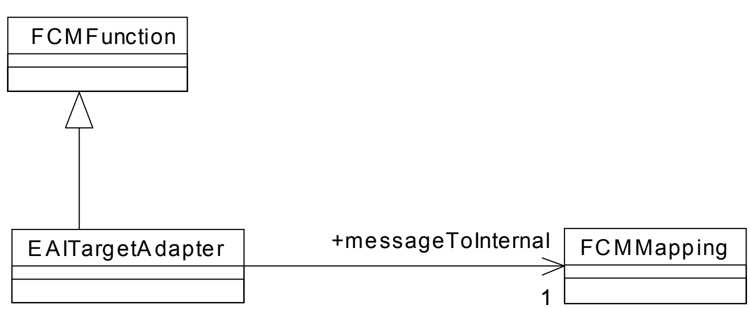



*Figure 6-14* EAI Target Adapter

### *Constraints*

An EAICallAdapter has two input terminals, one of which is an FCMTerminal that is *not* an EAITerminal and the other of which is an EAITerminal with the name “handleReply.”

An EAICallAdapter has two input terminals, one of which is an FCMTerminal that is *not* an eAITerminal and the other of which is an EAITErminal with the name “request.”

The input parameters of the FCMFunction that EAITargetAdapter invokes must be EAIParameters (with associated EAIMessageContent).

An EAITargetAdapter has a single input terminal, which is an EAITerminal with the name “in.”

There is no constraint on whether the outputs and faults of the invokes FCMFunction are FCMParameters or EAIParameters. However, they are unlikely to have associated EAIMessageContent because adapters are used to link messaging to other (internal) interfaces.

## *6.3.11.3 EAICallAdapter*

An EAICallAdapter is invoked synchronously by an application that wishes to make use of a service made available via a server; the server accepts a request message and sends a response message back to the service requester. It has two input terminals:

- call”: an FCMTerminal that a requesting application can use to invoke the call adapter.
- handleReply”: an EAITerminal that handles a reply.

It has two output terminals:

- request”: the EAITerminal from which the request message is sent.
- out”: an FCMTerminal to which the reply message is mapped.

EAICallAdapter is a subclass of FCMFunction.

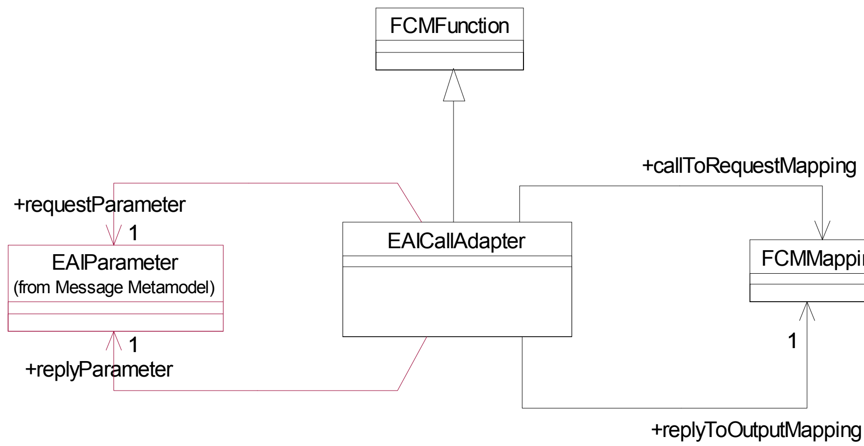



*Figure 6-15* EAICallAdapter

The call adapter has two mappings, one of which specifies how the call input parameters are mapped to the request message; the other specifies how the return message is mapped to output parameters represented by the "out" terminal.

From the point of view of the requesting application, the EAICallAdapter is a single FCMFunction that takes an input on its “call” terminal and produces an output on its “out” terminal. Within the EAI model, this function is realized as follows (i.e., this is effectively the behavior of the FCMOperation invoked by the function).

1. Map the “call” input into a request message using the callToRequestMapping.
2. Place the request message on the “request” output terminal.
3. Wait for a reply message to be received on the “handleReply” input terminal.
4. Map the reply message to an output value using the replyToOutput mapping.
5. Place the output value on the “out” terminal.

### *Constraints*

The parameter associated with the “out” terminal of an EAICallAdapter must be an EAIParameter with a message that is an EAIRequestReplyAdapter.

The FCMOperation invoked by an EAICallAdapter (when considered as an FCMFunction, see Figure 6-2) must have exactly one input FCMParameter and exactly one output FCMParameter.

The input FCMTerminal of an EAICallAdapter (that is *not* an EAITerminal) is associated with the input FCMParameter and the output FCMTerminal (that is *not* an EAITerminal) is associated with the output FCMParameter.

### 6.3.11.4 EAIRequestFormat

EAIRequestFormat is a subclass of EAIMessageContent that is used to specify a request message that may be produced by an EAICallAdapter and received by an EAIRequestReplyAdapter. While the structure of an EAIRequestFormat is just like any other EAIMessageContent, a request message has the added semantic responsibility of identifying the terminal to which a reply to the message should be sent. How this identification is made is not explicitly defined in the metamodel syntax for an EAIRequestFormat, but it must be computable from the information specified for a request message (e.g., some sort of unique identifier for a reply terminal might be included in a header part of the message or some other sort of language element might be modeled to provide a logical identification of a terminal).

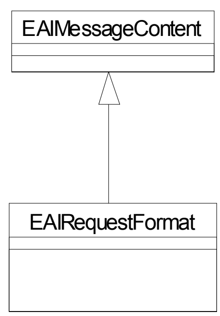



*Figure 6-16* EAIRequestFormat

### 6.3.11.5 EAIRequestReplyAdapter

An EAIRequestReplyAdapter is used to synchronously invoke a function of a server application. It has two input terminals:

- “requestIn”: an EAITerminal that accepts a message whose content is specified by an EAIRequestFormat (and thus provides some means of identifying a reply terminal).
- “handleReturn”: an FCMTerminal that receives the reply from the server application.

It has two output terminals:

- “replyOut”: the EAITerminal from which the reply message is sent.
- “call”: an FCMTerminal to which the request is mapped to be sent to the server application.

The request reply adapter has two mappings, one of which specifies how the “requestIn” input data are mapped to the server application call; the other specifies how the return data are mapped to the output message represented by the “replyOut” terminal.

From the point of view of the EAIModel, the EAIRequestReplyAdapter is a single FCMFunction that takes a request message on its "requestIn" terminal and produces a reply message on its "replyOut" terminal. This function is realized as follows (i.e., this is effectively the behavior of the FCMOperation invoked by the function).

1. Map the "requestIn" message into the data required for the server application call using the requestToCallMapping.
2. Place the call data on the "call" output terminal.
3. Wait for return data to be received on the "handleReturn" input terminal.
4. Map the return data to a reply message using the returnToReply mapping.
5. Place the reply message on the "replyOut" terminal and transmit it to the reply terminal identified in the request message.

Note that, in addition to simply being placed on the "replyOut" terminal, the reply message is transmitted to the reply terminal that is dynamically identified by the incoming request message. Request messages are generated by EAICallAdapters, with the reply terminal of the request message being the "handleReply" terminal of the EAUICallAdapter. Thus, the semantics of an EAIRequestReplyAdapter effectively results in the creation of a dynamic and temporary EAILink between the "replyOut" terminal of the EAIRequestReplyAdapter and the "handleReply" terminal of the EAICallAdapter that generated the request message.

Now, if the identified reply terminal is *not* an EAIQueuedInputTerminal, then the dynamic EAILink is considered to have *synchronization* = *unspecified*. The reply message is simply placed on the identified input terminal. However, if the identified reply terminal *is* an EAIQueuedInputTerminal (see Section 6.3.8, "EAIQueuedInputTerminal and EAIQueuedOutputTerminal," on page 6-10), then the dynamic EAILink is considered to have *synchronization* = *asynchronous* and the reply message is placed on the inputQueue of the reply terminal.

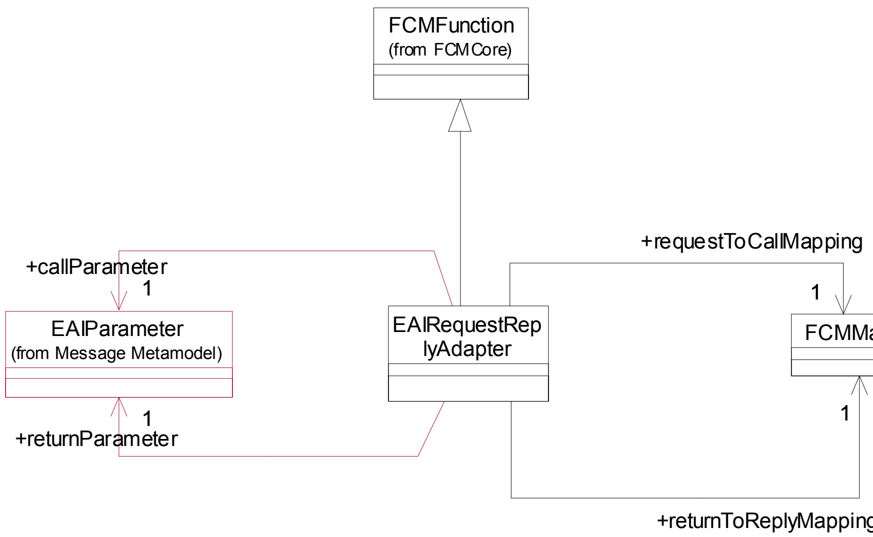



*Figure 6-17* EAIRequestReplyAdapter

### *Constraints*

An EAIRequestReplyAdapter has two input terminals, one of which is an FCMTerminal that is *not* an EAITerminal and the other of which is an EAITerminal with the name “requestIn.”

An EAIRequestReplyAdapter has two output terminals, one of which is an FCMTerminal that is *not* an EAITerminal and the other of which is an EAITerminal with the name “replyOut.”

The FCMOperation invoked by an EAIRequestReplyAdapter (when considered as an FCMFunction, see Figure 6-2) must be an EAIOperation with exactly one input EAIParameter, with a message that is an EAIRequestFormat, and exactly one output EAIParameter.

The “requestIn” terminal of an EAIRequestReplyAdapter is associated with the input EAIParameter and the “replyOut” terminal is associated with the output EAIParameter.

The representation of the callParameter of an EAIRequestReplyAdapter is the output FCMTerminal and the representation of the returnParameter of an EAIRequestReplyAdapter is the “return” terminal. (The representation association for an FCMParameter is shown on Figure 6-6.)

# 6.4 Kinds of Operator

## 6.4.1 Operators

We define several specializations of EAIPrimitiveOperator and EAICompoundOperator. EAICompoundOperators combine more than one of the primitive EAI concepts represented by the PrimitiveOperators. Implementations of them do not need to follow this internal representation, provided that they obey the signature (in terms of the messages they receive and send) and the documented semantics.

### 6.4.1.1 EAIFilter

An EAIFilter is a subclass of EAIPrimitiveOperator.

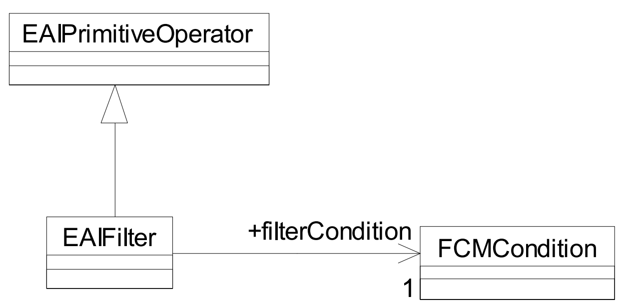



*Figure 6-18* Filter

A filter has one input terminal and two output terminals. The output terminals must be named “true” and “false.” If the message on the input terminal satisfies the filter condition, then it is copied to the output terminal named “true.” Otherwise, the message is copied to the output terminal named “false.”

### 6.4.1.2 EAIStream

EAIStream is an operator that allows 'quality of service' on a communication channel to be expressed.

The flow of control and data via EAILink between EAITerminals assumes that messages are always received in the order that they are sent and that there is basically no delay in their transmission.

In some implementations, a stream of messages may be received in a different order from that in which they are sent, and they may be received at a different rate from that at which they are sent. An EAIStream operator can be used to model this.

An EAIStream can be used to model reordering of incoming messages by maintaining a buffer.

In an implementation, an incoming message may be added to the buffer in a place determined by the streaming algorithm. An outgoing message may be sent at the same or different time as an incoming message is received. The streaming algorithm determines when to place messages from the top of the buffer onto the "out" terminal. Typically, this will be when the buffer contains a sufficient block of messages in the correct order.

All of this behavior is abstracted via an emissionCondition that determines under what circumstances a message is emitted from the stream. The message emitted may be any element of the buffer. Once emitted from the stream, the message is removed from the buffer.

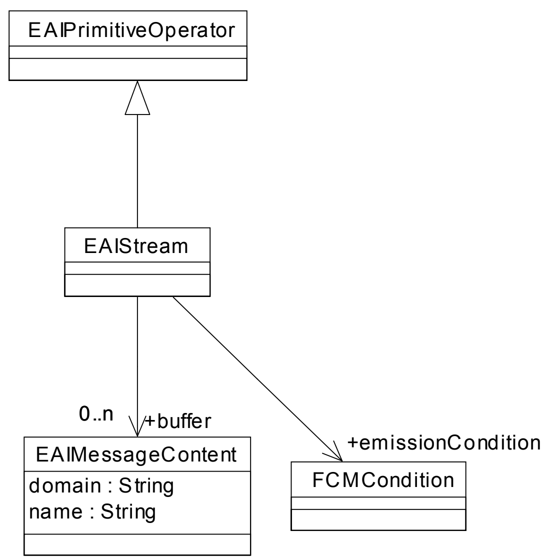



*Figure 6-19* Stream

### *6.4.1.3 EAIPostDater*

EAIPostDater is a subclass of EAIStream with a single input terminal ("in") and a single output terminal ("out").

On receipt of a message at its input terminal, it adds the message to the buffer, and creates an individual timingCondition for it. The timingCondition may entail a derivation from the content of the input message by a timerMapping. EAIPostDater holds the message until its individual timing condition is met, then emits it from its "out" terminal.

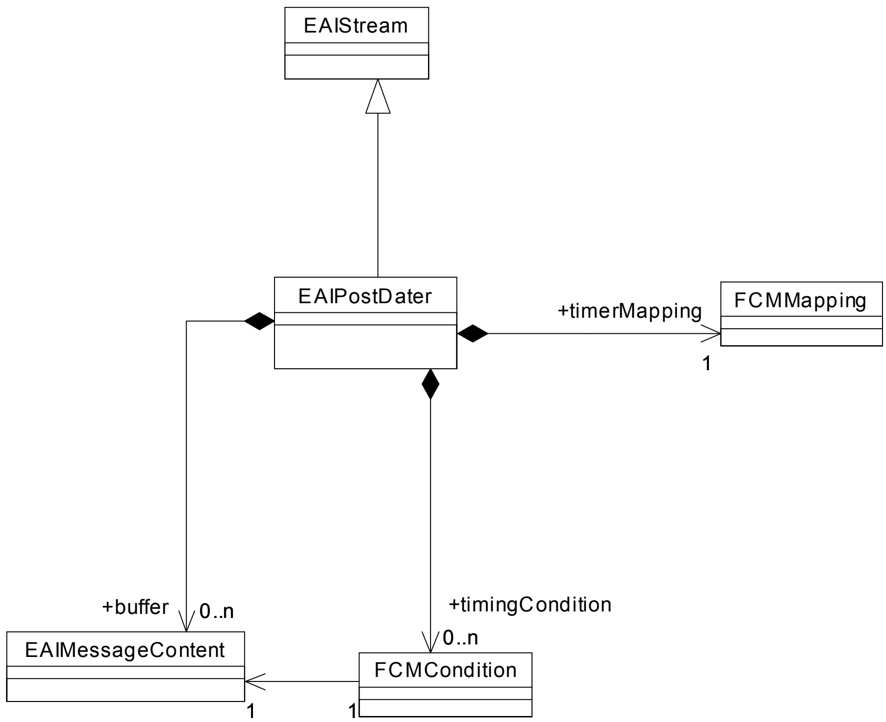



*Figure 6-20* EAIPostDater

## *6.4.1.4 EAITransformer*

A Transformer is a subclass of PrimitiveOperator with a single input terminal and a single output terminal.

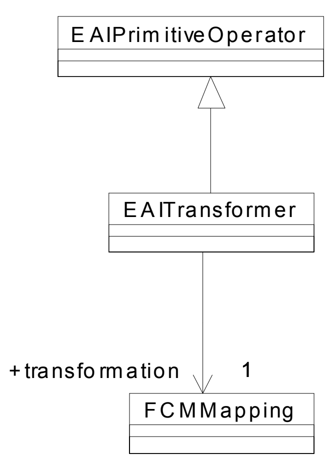



*Figure 6-21* Transformer

The output message is a transformation of the input message, as dictated by the transformation FCMMapping.

### 6.4.1.5 EAIDBTransformer

An EAIDBTransformer is a subclass of EAITransformer that has access to an EAIDatabase.

EAIDatabase is modeled as a subclass of EAIResource and has the property databaseName. Subclasses of EAIDatabase may specify further properties such as information required to connect to the database.

An EAIDBTransformer is an EAITransformer, which is itself an EAIPrimitiveOperator, which may have resources attached to it (see Section 6.3.10.4, “EAIPrimitiveOperator,” on page 6-14). An EAIDBTransformer is specifically required to have exactly one such resource, which must be an EAIDatabase.

Access to a database as a resource allows the transformation to make use of information contained in the database. In particular, it allows the message to be augmented (or enriched) with data from the database.

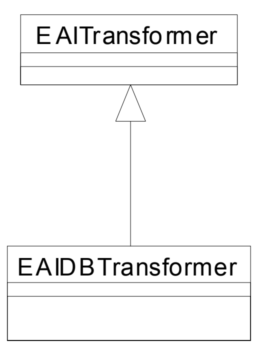



*Figure 6-22* EAIDBTransformer

#### *Constraints*

An EAIDBTransformer has exactly one resource, which is an EAIDatabase.

### 6.4.1.6 EAIAggregator

An EAIAggregator is a subclass of PrimitiveOperator. It has a single input terminal (“in”) and a single output terminal (“out”). Its purpose is to combine several messages (comprising an aggregate) into a single output message (EAIMessageAggregation). It is commonly used in conjunction with EAITimer, which can check for deadlines.

On receipt of a message, if there are no existing message aggregates, the aggregator creates one and adds the message to it.

On receipt of a subsequent message, the aggregator examines each existing aggregate, evaluating the addToAggregate condition (which will depend on the message header or body contents). If an aggregate exists for which addToAggregate evaluates to true, then the message is added to it.

Each time a message is added to an aggregate, the aggregateComplete condition is evaluated. If it evaluates to true, then a message is constructed from the messages it holds and is sent on the output terminal. The mapping from the messages contained in the aggregate to the message sent is specified by the aggregationMapping.

If the aggregateComplete condition does not evaluate to true, then no message is sent.

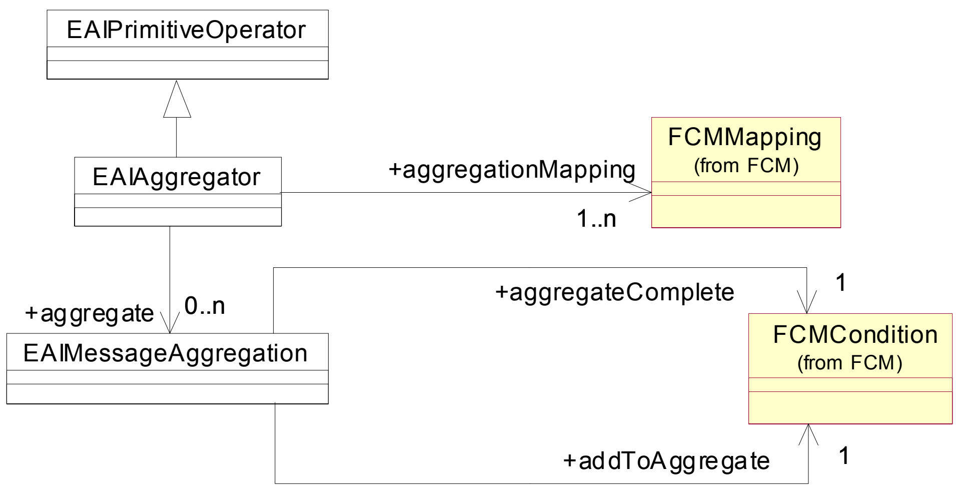



*Figure 6-23* EAIAggregator

### 6.4.1.7 *EAIRouterUpdate and EAIRouter*

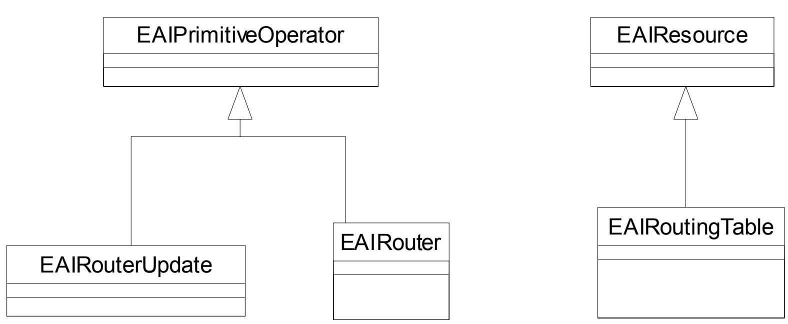



*Figure 6-24* EAIRouter and EAIRouterUpdate

An EAIRouter routes a message to destinations listed in an EAIRoutingTable, which is maintained by EAIRouterUpdate. An EAIRoutingTable is a kind of EAIResource. An EAIRouter and an EAIRouterUpdate must each be associated with a single resource, which is an EAIRoutingTable.

An EAIRouter is a primitive operator with a single input terminal ("in") and a single output terminal ("out"). The target terminals of any EAILinks connected to the output terminal of an EAIRouter are added to the EAIRoutingTable for that EAIRouter as the initial set of routing targets. This set may be changed by the operation of an EAIRouterUpdate operator. When a message is received on the input terminal of an EAIRouter, dynamic EAILinks are established between the output terminal of the EAIRouter and each of the terminals in the current set of routing targets of the EAIRoutingTable of the EAIRouter. The input message is then copied to the output terminal and thus sent to each of the routing targets.

An EAIRouterUpdate is a primitive operator with a single input terminal ("control") and no output terminals. It expects to receive a message that conforms to the EAIRouterUpdateFormat content type. Such a message can specify either the addition (adds) or removal (removes) of a single terminal from the routing table that is associated with the operator as a resource.

### 6.4.1.8 *EAISubscriptionOperator*

An EAISubscriptionOperator is a subclass of EAIPrimitiveOperator with a single input terminal ("subscribe") and no output terminals. It expects an EAISubscriptionFormat as input. On receipt of an EAISubscriptionFormat, it adds information on the specified subscription to an EAISubscriptionTable.

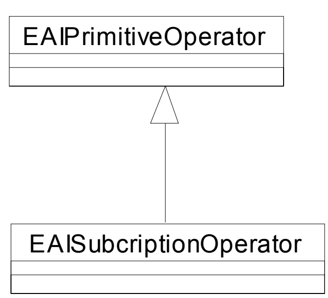



*Figure 6-25* SubscriptionOperator

A message that conforms to the EAISubscriptionFormat specifies a target EAITerminal and a set of EAISubscriptionRules. In Figure 6-26, this is shown as a pair of derived associations. This indicates that the target and associated subscription rules can be computed from the message content.

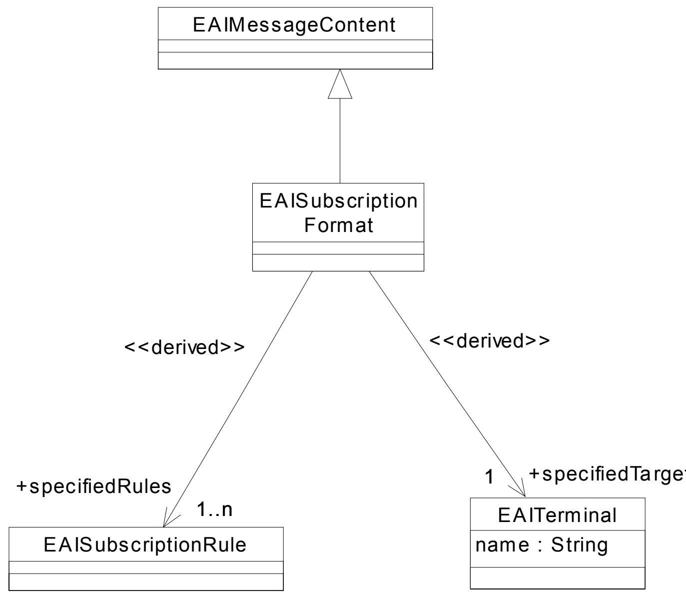



*Figure 6-26* EAISubscriptionFormat

An EAISubscriptionTable is an EAIResource that is used to record the subscriptions received by an EAISubscriptionOperator. An EAISubscriptionOperator is an EAIPrimitiveOperator, which may have attached resources (see Figure 6-25). An EAISubscriptionOperator is specifically required to have exactly one resource, which must be an EAISubscriptionTable. An EAIPublicationOperator (see Section 6.4.1.9, "EAIPublicationOperator," on page 6-30) referencing the same EAISubscriptionTable may then forward to subscribed target terminals messages that satisfy the subscription rules for those terminals.

An EAISubscriptionFilter is a subclass of EAIFilter. Its filterCondition is a set of EAISubscriptionRule.

### *Constraints*

An EAISubscriptionOperator has exactly one terminal, which is an input EAITerminal with the name "subscribe."

The input terminal of an EAISubscriptionOperator is associated with an EAIParameter that has a message that is an EAISubscriptionFormat.

An EAISubscriptionOperator must have exactly one resource, which is an EAISubscriptionTable.

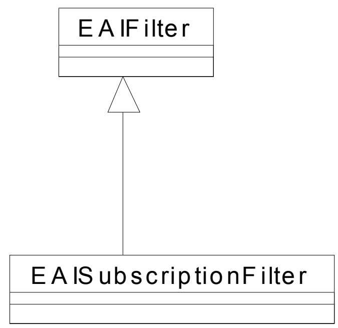



*Figure 6-27* SubscriptionFilter

An EAISubscriptionRule has subclasses EAITopicRule and EAIContentRule. An EAITopicRule tests whether a message was published to one or more of an allowed set of topics, as recorded in the header for that message (see also Section 6.4.2.3, “Relationship between topic-based publishers and subscribers,” on page 6-36). An EAIContentRule is a predicate that operates on the content of a message.

The filterCondition of an EASubscriptionFilter is an EAISubscriptionRule.

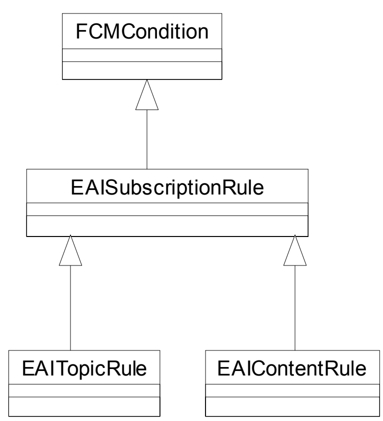



*Figure 6-28* EAISubscriptionRule, EAITopicRule, and EAIContentRule

### *6.4.1.9 EAIPublicationOperator*

#### ***Description***

The EAIPublicationOperator is used to model the publishing portion of the publish/subscribe more of information sharing. It forwards messages to target terminals recorded in the EAISubscriptionTable attached to it as a resource, if the messages meet the relevant subscription rules.

An EAIPublicationOperator is an EAIPrimitiveOperator with a single input terminal ("in") and a single output terminal ("out"). When a message arrives at the input terminal, the EAISubscriptionRules for all subscriptions in the current state of the EAISubscriptionTable are evaluated on the message. For each subscription for which the rule is true, a dynamic, temporary EAILink is effectively established from the output terminal to the subscriber EAITerminal from the subscription. The input message is then copied to the output terminal and thus distributed to each subscriber.

If the target terminal of a dynamic EAILink is *not* an EAIQueuedInputTerminal, then the dynamic EAILink is considered to have *synchronization = unspecified*. The published message is simply placed on the identified target terminal. However, if the identified target terminal *is* an EAIQueuedInputTerminal (see Section 6.3.8, "EAIQueuedInputTerminal and EAIQueuedOutputTerminal," on page 6-10), then the dynamic EAILink is considered to have *synchronization = asynchronous* and the published message is placed on the inputQueue of the target terminal.

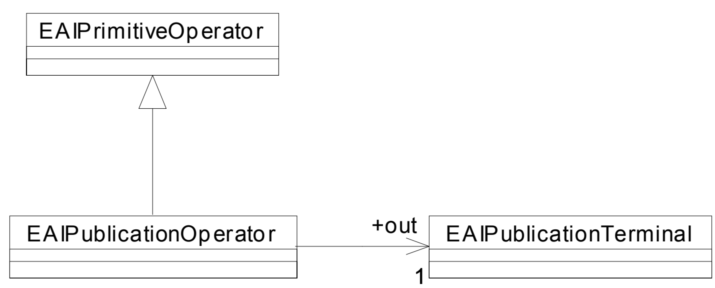



*Figure 6-29* EAIPublicationOperator and EAISubscriptionOperator

### *Constraints*

An EAIPublicationOperator has exactly one input terminal, which is an EAITerminal with the name "in," and exactly one output terminal, which is an EAITerminal with the name "out."

The messages of the EAIParameters associated with the two terminals of an EAIPublicationOperator must be the same.

An EAIPublicationOperator must have exactly one resource, which is an EAISubscriptionTable.

An EAIPublicationOperator has exactly one input terminal, which is an EAITerminal with the name "in," and exactly one output terminal, which is an EAITerminal with the name "out."

The messages of the EAIParameters associated with the two terminals of an EAIPublicationOperator must be the same.

### 6.4.1.10 EAITimeSetOperator

The TimeSetOperator is a subclass of EAIPrimitiveOperator, with a single input terminal ("set") and no output terminals. On receipt of a message, which must be specified by an EAITimerConditionFormat, it adds the timer and message applicability conditions given by the message to the list of conditions stored in the EAITimerConditionTable that is attached to it as a resource.

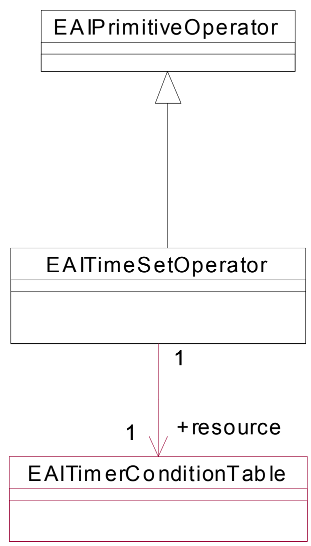



*Figure 6-30* TimeSetOperator

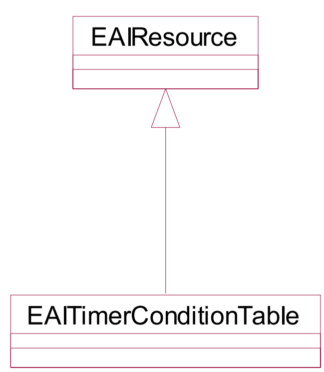



*Figure 6-31* TimerConditionTable

A message in EAITimerConditionFormat is composed of two FCMConditions:

- timerCondition specifies a deadline (a time constraint). This may be relative or absolute.

- correlationCondition specifies the messages to which the timerCondition applies. This is often a condition on an element of a message header, such as the commonly used 'correlation identifier.'

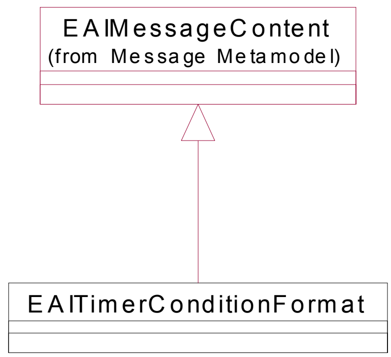



*Figure 6-32* EAITimerConditionFormat

#### *Constraints*

Messages received on the "set" terminal must be in EAITimerConditionFormat.

### *6.4.1.11 EAITimeCheckOperator*

EAITimeCheckOperator is a subclass of EAIPrimitiveOperator with a single input terminal ("check") and three output terminals ("ontime," "expiry," and "late"). On receipt of a message, it examines its set of conditions stored in the EAITimerConditionTable that is attached to it as a resource. If there is a timer condition that applies to the message, it checks that the condition is actually met. If so, the message is passed to the "ontime" terminal; if not, it is passed to the "late" terminal.

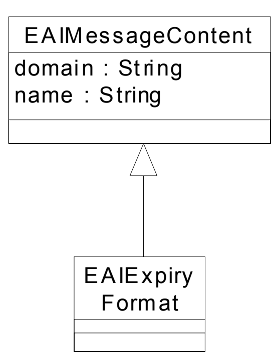



*Figure 6-33* EAIExpiryFormat

At the time that a particular timer condition expires, a message of format EAIExpiryFormat is sent from the "expiry" terminal.

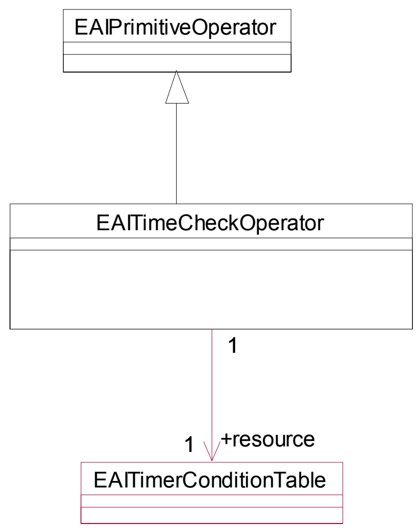



*Figure 6-34* EAITimeCheckOperator

### *6.4.1.12 EAITimer*

EAITimer is formed from a composition of EAITimeSetOperator and EAITimeCheckOperator.

It has two input terminals, "set" and "check" and the output terminals "out," "expiry," and "late" all of which map to terminals of the same name owned by the two primitive operators. Consequently, the "set" terminal causes the EAITimeSetOperator to be invoked, while messages sent to the "check" terminal cause the EAITimeCheckOperator to be invoked.

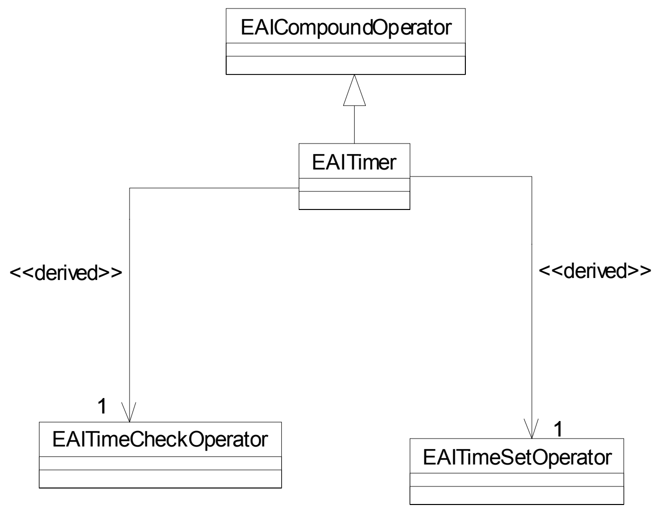



*Figure 6-35* EAITimer

***Constraints***

The instance of EAITimeCheckOperator and EAITimeSetOperator from which an EAITimer is formed share the same EAITimerConditionTable.

## *6.4.2 Topic-based publish/subscribe*

### *6.4.2.1 EAITopicPublisher*

An EAITopicPublisher is a subclass of EAISource. It sends messages for publication to an EAIPublicationOperator. The set of topics that it publishes messages on is denoted by publishesOn. This is a derived association, since a topic publisher need not declare the set of topic it publishes on.

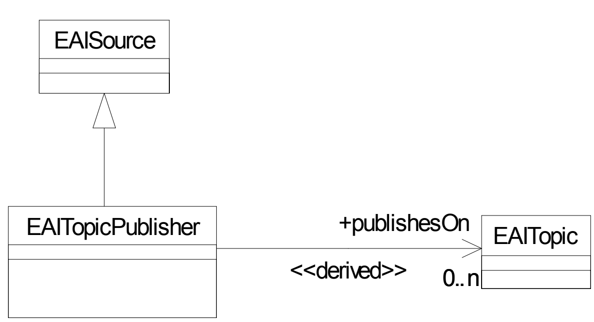



*Figure 6-36* EAITopicPublisher

### 6.4.2.2 Topics 'allowed' by an EAITopicRule

An abstract representation of an EAITopicRule is the set of Topics that it allows.

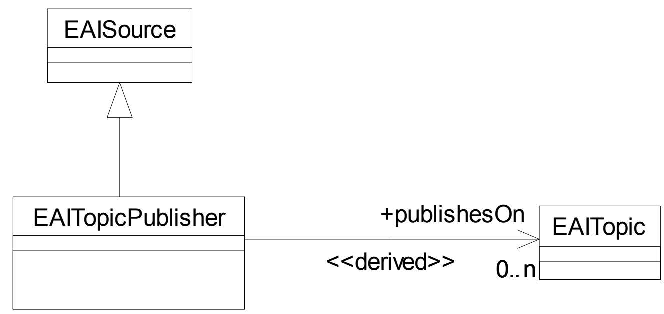



*Figure 6-37* Topics allowed by an EAITopicRule

### 6.4.2.3 Relationship between topic-based publishers and subscribers

Topic-based publishers and subscribers are related to each other via the topics that they produce and consume.

For an input terminal representing a subscriber connected to a particular PublicationOperator, the set of topics it is interested in (subscribesTo) is determined by the topic that its filterCondition allows.

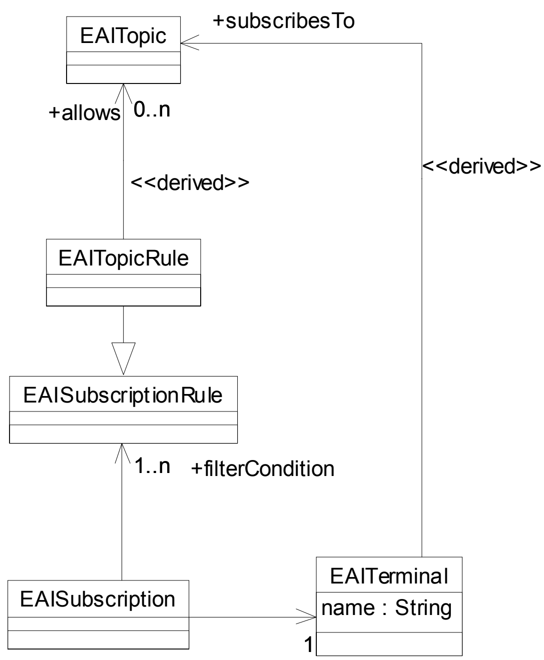



*Figure 6-38* Relationship between a terminal and the topics for which it has a subscription

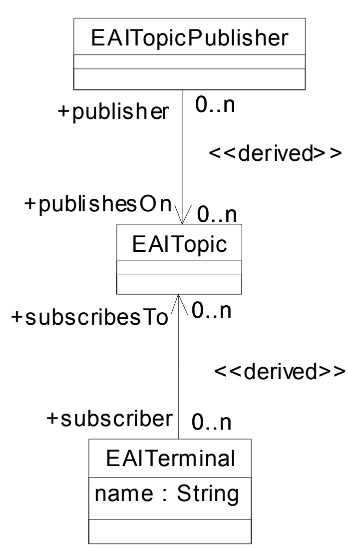



*Figure 6-39* Relationship between publishers, subscribers and topics

# 6.5 CCA Component Library for EAI

This section specifies the CCA component library for EAI and mapping between EAI and CCA concepts. CCA provides for the modeling of collaboration similar to the EAI models in Chapters 7 and 8. The component library specifies the set of components required in CCA to represent the same concepts as the EAI meta model. By providing this component library and mapping between EAI and CCA users may transform models between EAI and CCA tools, integrating EAI systems with collaborations modeled with CCA. This information may be used by EAI or CCA tool vendors to automate such transformation and integration or may be used directly by users in a manual process.

For each of the listed EAI model elements a corresponding library component is defined. In each case the library component has the same name as the corresponding EAI model element.

## 6.5.1 Operators

### 6.5.1.1 EAIPrimitiveOperator

EAIPrimitiveOperator corresponds to an unconstrained CCA ProcessComponent.

The Terminal of the EAIPrimitiveOperator corresponds to Port of the CCA ProcessComponent.

Input Terminal corresponds to a CCA FlowPort with metaattribute direction = responds.

Output Terminal corresponds to a CCA FlowPort with metaattribute direction = initiates.

The handled ContentFormat of a Terminal in the EAIPrimitiveOperator corresponds to the type DataElement of the CCA FlowPort.

The Choreography of the CCA ProcessComponent corresponding to an EAIPrimitiveOperator will have CCA PortActivity. This represents each CCA FlowPort corresponding to EAI input Terminal, followed by CCA Transition with target on CCA PortActivity that represents each CCA FlowPort corresponding to EAI output Terminal.

A CCA ProcessComponent, corresponding to an EAIPrimitiveOperator, can be utilized in a CCA Composition as a CCA ComponentUsage that uses the CCA ProcessComponent. For each CCA Port in the CCA ProcessComponent, there will be a CCA PortConnector corresponding to the CCA FlowPort of the used ProcessComponent.

In CCA, there is no fundamental distinction between primitive and non-primitive ProcessComponents. Rather, the “primitiveness” of a ProcessComponent is not externally observable. The CCA ProcessComponent may optionally have internal Composition detail, using other ProcessComponents.

In CCA, there is no fundamental distinction between primitive and non-primitive ProcessComponents. Rather, the “primitiveness” of a ProcessComponent is not externally observable. The CCA ProcessComponent may optionally have internal Composition detail, using other ProcessComponents.

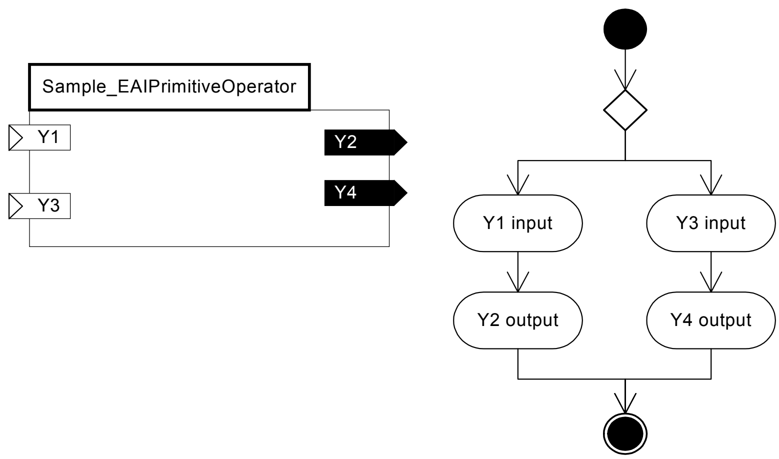



*Figure 6-40* CCA notation for a sample generic EAIPrimitiveOperator

### 6.5.1.2 *EAITransformer*

EAITransformer is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with one CCA FlowPort with direction = responds and one CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAITransformer will show a CCA PortActivity on the FlowPort with direction = responds, followed by a CCA PortActivity on the FlowPort with direction = initiates.

The input and output CCA FlowPort will have different DataElement types. The ProcessComponent will transform from the input DataElement type to the output DataElement type.

The transformation to be performed on the DataElement contents can be specified in a Property of the CCA ProcessComponent as an expression, script, or transformation specification in any of the transformation languages available. Alternatively, the transformation can be delegated into usages of other technology-specific transformation ProcessComponents in the internal Composition or into EAI transformer implementations.

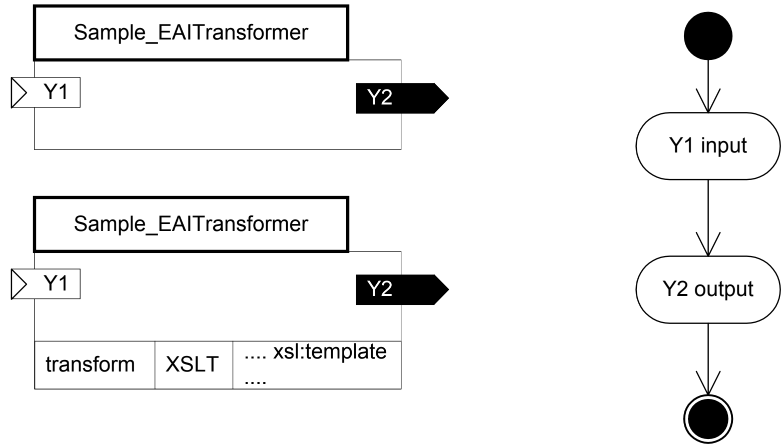



*Figure 6-41* CCA notation for sample EAITransformer

### *6.5.1.3 EAIFilter*

EAIFilter is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with one CCA FlowPort with direction = responds and two CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAIFilter will show a CCA PortActivity on the FlowPort with direction = responds, followed by a choice vertex, followed by a CCA PortActivity on each of the FlowPort with direction = initiates.

The input and each output CCA FlowPort will have the same DataElement type.

The criteria for the choice of true or false output terminal Port can be specified in a Property of the CCA ProcessComponent as an expression in any of the languages available. Criteria logic can also be delegated into usages of other ProcessComponents in the internal Composition.

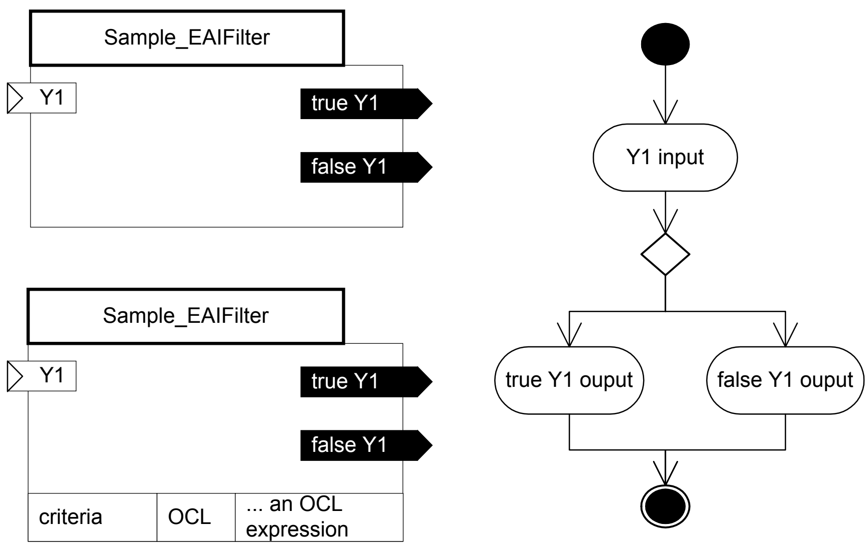



*Figure 6-42* CCA notation for a sample EAIFilter

### 6.5.1.4 *EAIStream*

EAIStream is a specialized EAIPrimitiveOperator. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds and a single CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAIStream will show a CCA PortActivity on the FlowPort with direction = responds, followed by a Fork, followed by CCA PortActivity on the FlowPort with direction = initiates, followed by a Join.

The input and output CCA FlowPort will have the same DataElement type. The ProcessComponent will store inputs to be sent later, possibly in a different order, through the output terminal FlowPort.

The algorithm used to determine when, and in which order, the incoming messages will be posted in the output terminal FlowPort can be specified as a Property of the EAIStream component, or it can be delegated into usages or other ProcessComponents in the internal Composition.

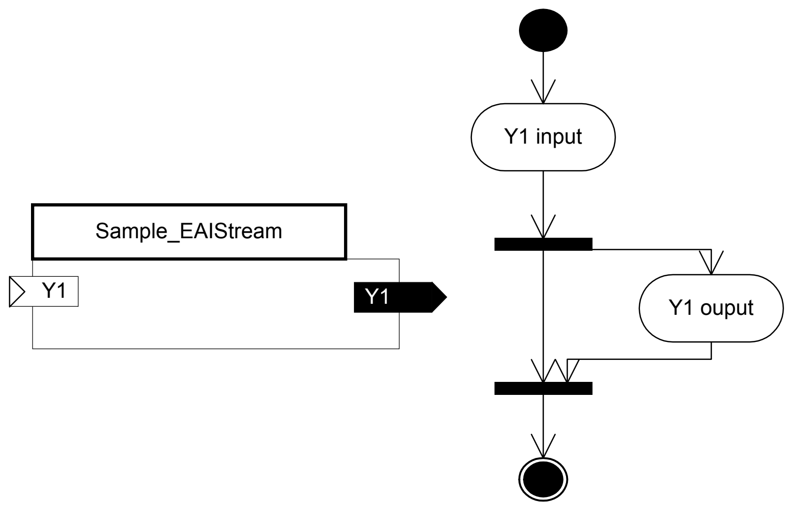



*Figure 6-43* CCA notation for a sample EAIStream

### *6.5.1.5 EAICompoundOperator*

EAICompoundOperator corresponds to an unconstrained CCA component. It will use other EAI Operator or Adapter ProcessComponents in the internal Composition.

The ProcessComponent for EAICompoundOperator will have externally connectable Ports that will be delegated into Ports of the internally used ProcessComponent.

Incoming messages on the external Port of the EAICompoundOperator ProcessComponent will be delivered to the internally connected Port of the ProcessComponent operators and adapters used.

Outgoing messages from the internally connected Port of the used ProcessComponent operators and adapters will be forwarded to the external outgoing Port of the EAICompoundOperator ProcessComponent.

This recursive composition capability of CCA corresponds to FCM and EAI recursive composition of nodes, operators, and adapters.

For the user of an EAICompoundOperator ProcessComponent, there is no difference between using a Compound or a Primitive Operator. The internal composition of the Compound Operator remains encapsulated by the ProcessComponent. The user can only observe the external Port and Choreography of the ProcessComponent.

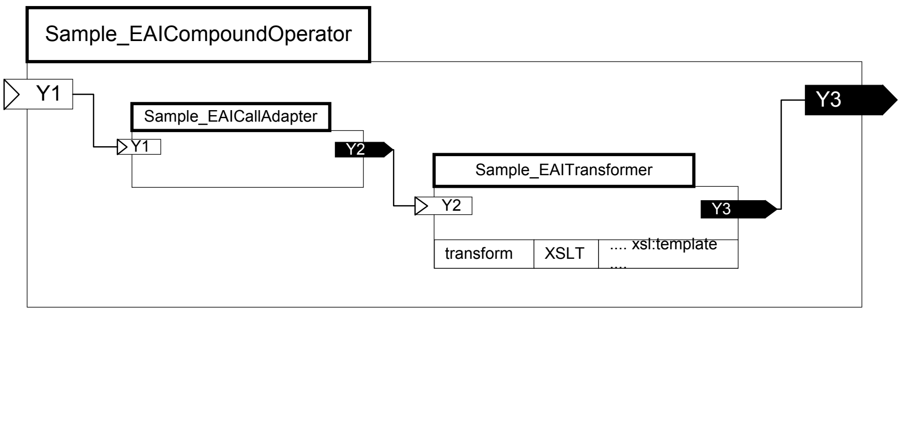



*Figure 6-44* CCA notation for sample EAICompoundOperator

## 6.5.2 Adapters

### 6.5.2.1 EAISourceAdapter

EAISourceAdapter is a specialized FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = initiates.

The Choreography of the CCA ProcessComponent corresponding to an EAISourceAdapter will show a CCA PortActivity on the FlowPort with direction = initiates.

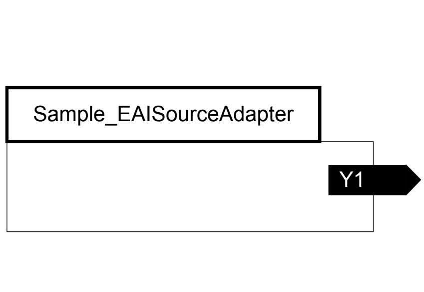



*Figure 6-45* CCA notation for a sample EAISourceAdapter

When the EAISourceAdapter is to be utilized in Pull mode, an additional FlowPort will respond to a generic “Get” message that will trigger retrieval from the system and initiate the output.

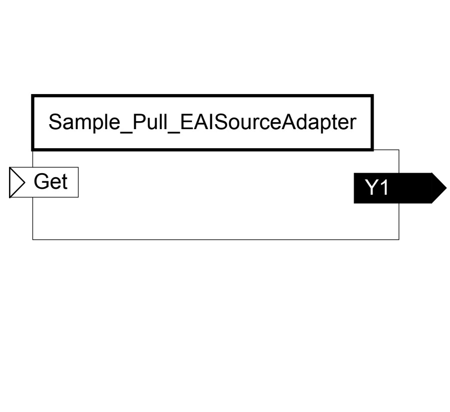



*Figure 6-46* CCA notation for a sample Pull mode EAISourceAdapter

### *6.5.2.2 EAITargetAdapter*

EAISourceAdapter is a specialized FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds.

The Choreography of the CCA ProcessComponent corresponding to an EAITargetAdapter will show a CCA PortActivity on the FlowPort with direction = responds.

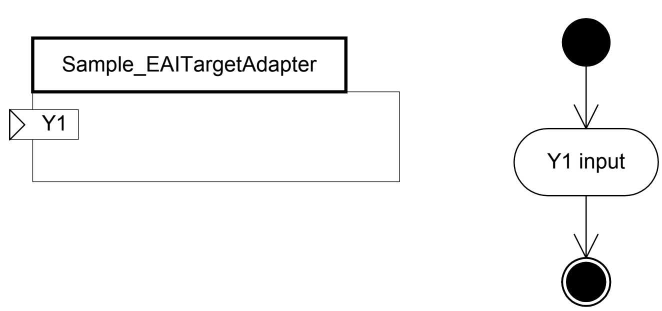



*Figure 6-47* CCA notation for a sample EAITargetAdapter

### 6.5.2.3 EAIQueuedTargetAdapter

EAISourceAdapter is a specialized FCMFunction. It corresponds to a CCA ProcessComponent with a single CCA FlowPort with direction = responds.

An EAIQueuedTargetAdapter offers the same externally observable contract as the EAITargetAdapter but with different internal behavior, namely, queued delivery of messages to the system.

Queueing of messages can be directly implemented or delegated into usages of technology-specific message-queue ProcessComponents in the internal composition.

### 6.5.2.4 EAICallAdapter

EAISourceAdapter is a specialized FCMFunction. It corresponds to a CCA ProcessComponent with a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates.

Alternatively, an EAICallAdapter may correspond to a CCA ProcessComponent with a ProtocolPort, with subPorts obeying a Protocol having a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates. This aggregation in a single ProtocolPort of the FlowPorts for the call and response messages provides a single connection point for the full call-response, which is similar to the conventional functional invocation in programming languages.

The Choreography of the CCA ProcessComponent corresponding to an EAICallAdapter will show a CCA PortActivity on the FlowPort with direction = responds, followed by a CCA PortActivity on the FlowPort with direction = initiates.

An EAICallAdapter accepts synchronous calls that are not externally observable. It converts these to asynchronous messages that are sent on the output terminal initiating FlowPort. It receives a response on the input terminal responding FlowPort and passes an equivalent response to the caller. The EAICallAdapter must implement the logic and mechanisms to wait for the asynchronous response and rebind to the thread of the calling process.

The input and output CCA FlowPort may have the same or different DataElement types. The ProcessComponent will convert the input to the type required by the system. The system will respond with information of a certain type that the ProcessComponent must convert into the output DataElement type.

The transformation to be performed on the DataElement contents can be specified in Properties of the CCA ProcessComponent as an expression, script, or transformation specification in any of transformation languages available. Alternatively, the transformation can be delegated into usages of other technology-specific transformation ProcessComponents in the internal Composition.

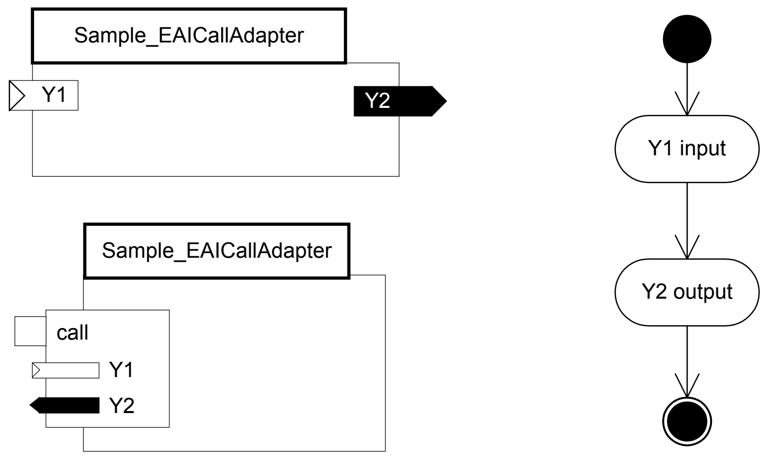



*Figure 6-48* CCA notation for sample EAICallAdapter

#### 6.5.2.5 *EAIRequestReplyAdapter*

EAISourceAdapter is a specialized FCMFunction. It corresponds to a CCA ProcessComponent with a CCA FlowPort with direction = responds and a CCA FlowPort with direction = initiates.

Externally, an EAIRequestReplyAdapter exposes similar contract and behaves like the EAICallAdapter.

The EAIRequestReplyAdapter accepts asynchronous messages. It invokes a system synchronously and returns the response as a message that other applications can process asynchronously. The RequestReplyAdapter presents an asynchronous interface on a synchronous invocation.

### 6.5.3 *CCA and EAI Metamodel Mapping Tables*

The following table shows the mapping between EAI and CCA model elements. In many cases the EAI library component is also part of the mapping.

*Table 6-1* Model elements mapping table

| **EAI metamodel element** | **CCA metamodel element** | **Library Component (Component Used)** |
|---|---|---|
| EAIFlow | ProcessComponent | |
| EAIRouterComposition | ProcessComponent | |
| EAIPrimitiveOperator | ComponentUsage | EAIPrimitiveOperator |
| EAICompoundOperator | ComponentUsage | EAICompoundOperator |
| EAITargetAdapter | ComponentUsage | EAITargetAdapter |

*Table 6-1* Model elements mapping table

| EAI metamodel element | CCA metamodel element | Library Component (Component Used) |
|---|---|---|
| EAISourceAdapter | ComponentUsage | EAISourceAdapter |
| EAICallAdapter | ComponentUsage | EAICallAdapter |
| EAIRequestReplyAdapter | ComponentUsage | EAIRequestReplyAdapter |
| EAIFilter | ComponentUsage | EAIFilter |
| EAIStream | ComponentUsage | EAIStream |
| EAIPostDater | ComponentUsage | EAIPostDater |
| EAITransformer | ComponentUsage | EAITransformer |
| EAIDBTransformer | ComponentUsage | EAIDBTransformer |
| EAIAggregator | ComponentUsage | EAIAggregator |
| EAIRouter | ComponentUsage | EAIRouter |
| EAIRouter | ComponentUsage | EAIRouter |
| EAIRouterUpdate | ComponentUsage | EAIRouterUpdate |
| EAISubscriptionOperator | ComponentUsage | EAISubscriptionOperator |
| EAISubscriptionFilter | ComponentUsage | EAISubscriptionFilter |
| EAIPublicationOperator | ComponentUsage | EAIPublicationOperator |
| EAITimeSetOperator | ComponentUsage | EAITimeSetOperator |
| EAITimeCheckOperator | ComponentUsage | EAITimeCheckOperator |
| EAITimer | ComponentUsage | EAITimer |
| EAISource | Port with direction = responds | |
| EAIQueuedSource | Port with direction = responds | |
| EAITopicPublisher | | |
| EAISink | Port with direction = initiates | |
| EAIQueuedSink | Port with direction = initiates | |
| EAILink | Connection | |
| EAIMessageOperation | FlowPort or OperationPort | |
| EAITerminal | PortConnector | |
| EAIQueuedInputTerminal | PortConnector | |
| EAIQueuedOutputTerminal | PortConnector | |
| EAIPublicationTerminal | | |
| EAISubscriptionRule | | |
| EAITopicRule | | |
| EAIContentRule | | |
| EAIMessageTimerCondition | | |

*Table 6-1* Model elements mapping table

| EAI metamodel element | CCA metamodel element | Library Component (Component Used) |
|---|---|---|
| EAIMessageContent | CompositeData | |
| EAIExceptionNotice | CompositeData | |
| EAIRequestFormat | | |
| EAIQueue | | |
| EAIContent | | |
| EAIRouterUpdateFormat | | |
| EAIAddTargetFormat | | |
| EAISubscriptionFormat | | |
| EAIResource | | |
| EAIMessageAggregation | | |
| EAISubscription | | |
| EAITopic | | |

Examples of the CCA modeling elements are presented in Chapter 11.

# EAI Common Application Metamodel 7

## 7.1 Business Requirements and Value

The current trend for new applications is to embrace open Web standards that simplify construction and scalability. As new applications are built, it is crucial to integrate seamlessly with existing systems while introducing new business models and new business processes.

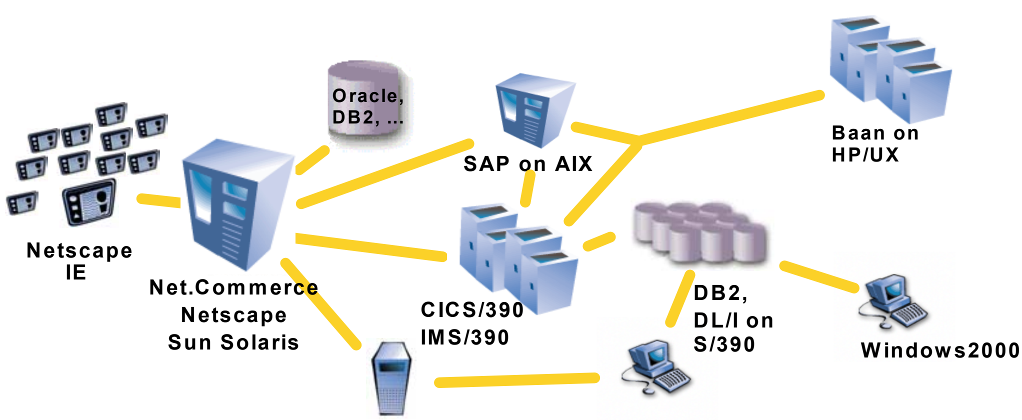



*Figure 7-1* Multiple Application and Development Environments

Analysts from the Meta Group estimate that more than 70% of corporate data lives on the mainframe, much of that on the S/390. Many transactions may be initiated by a Windows/NT or Unix server, but they will be completed on the mainframe under applications, such as CICS or IMS applications. It is important to leverage and reuse these existing assets, including stored procedures, to provide interoperability with existing applications.

The above figure depicts multiple application components with multiple development teams and environments. Where is the application in this picture? Everywhere! How is the application assembled? With connectors!

Connectors are a central part of the application framework for e-business. The demand is to connect to anything interesting as quickly, and as easily, as possible.

A connector is required to match the interface requirements of the adapter and the legacy application. It is also required to map between the two interfaces. Standardized metamodels for application interfaces allow reuse of information in multiple connector tools. It will not only reduce work to create a connector, but also reduce work needed to develop connector builder tools, thus an incentive to connector suppliers.

## 7.2 Common Application Metamodel for Applications Interfaces

Business integration technology requires connectors to provide interoperability with existing applications. Connectors support leveraging and reuse of data and business logic held within existing application systems. The job of a connector is to connect from one application system server “interface” to another; it is not meant for an individual application program. Therefore, an application-domain interface metamodel describes signatures for input and output parameters and return types for a given application system domain (e.g., IMS, MQSeries); it is not for a particular IMS or MQSeries application program. The metamodel contains both syntactic and semantic interface metadata.

The following figure showing the EAI metamodel for application interfaces enables integration of application components into event-based messaging model including Flow models.

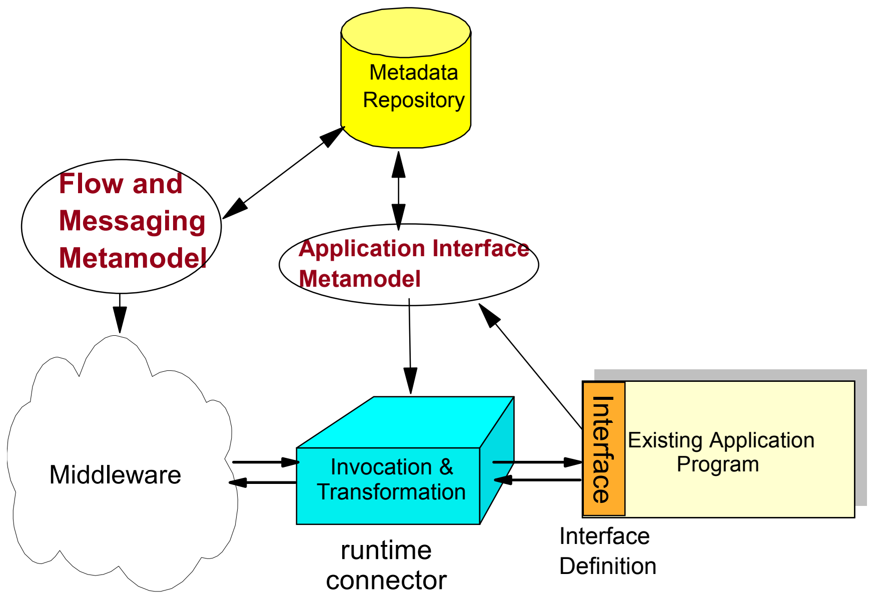



*Figure 7-2* Application Interface Metamodel

The flow and messaging middleware invokes applications through the application interfaces. These interfaces are the access points to the applications through which all input and output is connected to the middleware. The interfaces are described in terms of the Application Interface Metamodels. Transformation processing according to the metamodel could take place in source/client applications, target applications, or a gateway.

## *7.2.1 End-to-End Connector Usage Using EAI Common Application Metamodel*

The EAI Common Application Metamodel (CAM) consists of meta-definitions of message signatures, independent of any particular tool or middleware. Different connector builder tools can use this information to ensure the “handshaking” between these application programs, across different tools, languages, and middleware. For example, if you have to invoke an MQSeries application, you would need to build an MQ message using data from a GUI tool and deliver it using the MQ API. Similarly, when you receive a message from the MQSeries application, you would need to get the buffer from MQSeries, parse it and then put it into a GUI tool data structure. These functions can be designed and implemented efficiently by a connector builder tool using EAI CAM as standardized metamodels for application interfaces.

EAI CAM can be populated from many sources, including copy books, to generate HTML forms and JavaServer Page (JSP) for gathering inputs and returning outputs. An example of a connector as depicted in the previous figure is that the flow and message middleware makes a function call to an enterprise application by calling the connector that then calls the enterprise application API. The connector does language and data type mappings, for example, to translate between XML documents and COBOL input and output data structures based on EAI CAM. Connectors and EAI CAM provide the end-to-end integration between the middleware and the enterprise applications.

Using IMS as an example: Let's say that you must pass an account number to an IMS transaction application program from your desktop to withdraw $50.00. With EAI CAM and a connector builder tool, you will first generate an input HTML form and an output JSP; and develop a middleware code necessary to support the request. The desktop application fills the request data structure (i.e., an input HTML form) with values and calls the middleware. The middleware service code will take the data from the GUI tool, build an IMS Connect XML-formatted message, and deliver the message to the IMS gateway (i.e., IMS Connect) via TCP/IP. IMS Connect translates between the XML documents and the IMS message data structures in COBOL using the metadata definitions captured in EAI CAM. It then, in turn, sends the IMS message data structures to IMS via Open Transaction Manager Access (OTMA). The IMS COBOL application program runs, and returns the output message back to the middleware service code via IMS Connect. The middleware service code gets the message and populates the output JSP page (i.e., previously generated GUI tool reply data structures) with the reply data. The transaction output data will then be presented to the user.

IMS Connect and IMS OTMA are connector products that enable applications to interact with systems outside the host machine. For example, IBM Connect allows IMS to exchange data with sources outside z/Series using TCP/IP. IBM's WebSphere Studio is an example of a 'connector builder tool.' Once the connector builder tool has generated a servlet and/or transformer code for the application, the code can be deployed on a web server such as IBM WebSphere Application Server to communicate with the backend application via connectors such as IBM Connect and OMS OTMA. Below is a picture to help explain.

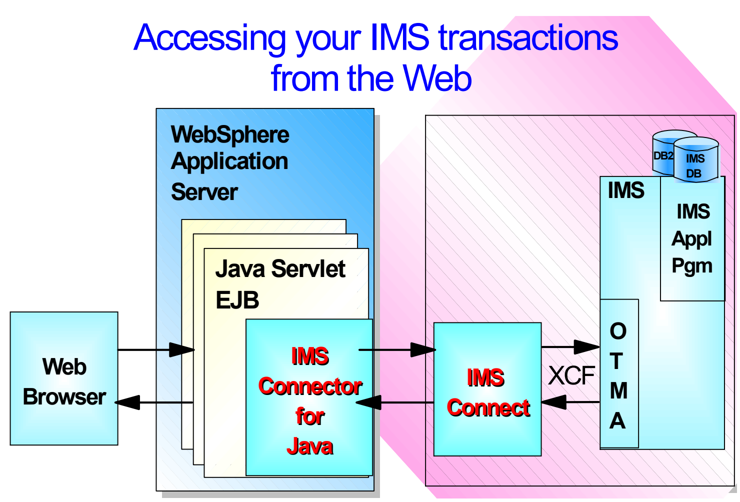



*Figure 7-3* Accessing IMS Transactions

## 7.3 Common Application Metamodel

CAM is a group of interface metamodels that consist of enterprise application interface metamodels, language metamodels, and physical representation metamodels. These include C, C++, Java, COBOL, PL/I, Type Descriptor, TDLang, IMS transaction messages, IMS MFS, and CICS BMS, etc. Note that the Java metamodel is defined in the OMG EDOC (Enterprise Distributed Object Computing) specification.

CAM is highly reusable and independent of any particular tool or middleware. CAM is an incentive to connector suppliers. It reduces work to create and develop connector and/or connector-builder tools. With CAM, tools can now easily access enterprise applications, e.g., IMS and CICS applications; and tools can also access any CAM enabled applications. CAM is used to describe information needed to easily integrate applications developed in common programming models with other systems. CAM can be used for both synchronous and asynchronous invocations.

Because CAM also provides physical representation of data types and storage mapping to support data transformation in an enterprise application integration environment, it enables Web services for enterprise applications.

In a nutshell, CAM is needed for

- Connector and/or connector-builder tools (Development time).
- Data transformation in an enterprise application integration environment (Execution time).
- Data type mapping between mixed languages.
- Data translations from one language and platform domain into another.
- Data driven impact analysis for application productivity and quality assurance.
- Viewing of programming language data declarations by developers.

CAM uses MOF and UML class modeling mechanisms. Every CAM class in an instance of a MOF class at the M2 level.

### 7.3.1 Enterprise Application Interface Metamodels

The Enterprise Application Interface metamodel describes signatures for input and output parameters and return types for application system domains.

The Enterprise Application Interface Metamodels listed as follows are non-normative and can be found in Appendix A.

- IMS Transaction Message
- IMS MFS
- IMS CICS BMS

## 7.3.2 Language Metamodels

The language metamodel (e.g., COBOL metamodel) is used by enterprise application programs to define data structures (semantics) that represent connector interfaces. An association between language metamodels (semantics) and the physical layout metamodel (syntactic) is necessary in order for the marshaller to correctly format the byte string. This association between language metamodels and Type Descriptor metamodel is further detailed in Section 7.3.9, "Physical Representation Model: TDLang Interaction Diagram," on page 7-23. It is important to connector developers that connector tools show the source language, the target language, and the mapping between the two languages. The CAM language metamodel also includes the declaration text in the model that is not editable (i.e., read-only model). Because the connector/adapter developer would probably prefer to see the entire COBOL data declaration, including comments and any other documentation that would help him/her understand the business role played by each field in the declaration.

The language metamodel is also to support data driven impact analysis for application productivity and quality assurance. (But, it is not the intention of the CAM to support reproduction of copybooks.)

The language metamodels describing application interface data are listed as follows:

- C
- C++
- COBOL
- PL/I
- Java (Java metamodel is in the OMG EDOC specification.)

These language metamodels are found in Chapter 13.

## 7.3.3 Physical Representation Model: TDLang Metamodel

The TDLang metamodel serves as base classes to CAM language metamodels by providing a layer of abstraction between the Type Descriptor metamodel and any CAM language metamodel, including higher level languages. All TDLang classes are abstract and common to all the CAM language metamodels. All associations between TDLang classes are marked as "volatile," "transient," or "derived" to reflect that the association is derived from the language metamodel. The TDLang model does not provide any function on its own, but it is the type target for the association from the Type Descriptor metamodel to the language metamodels.

With the TDLang base classes, the Type Descriptor metamodel can be used as a recipe for runtime data transformation (or marshaling) with the language-specific metamodel for overall data structures and field names, without duplicating the aggregation (parent-child) associations present in the language model.

The TDLang model eliminates the need to have unique associations from each language model to the Type Descriptor model (e.g., cobolToTD and cToTD). All language models can access InstanceTDBase by calling the instanceTDBase association through the parent TDLangElement class.

The following figure illustrates the TDLang Metamodel. TDLang connects language models to the Type Descriptor Model. The TDLang metamodel acts as a generic placeholder for a variety of language models to inherit from.

Following the diagram is a brief explanation of what each class represents.

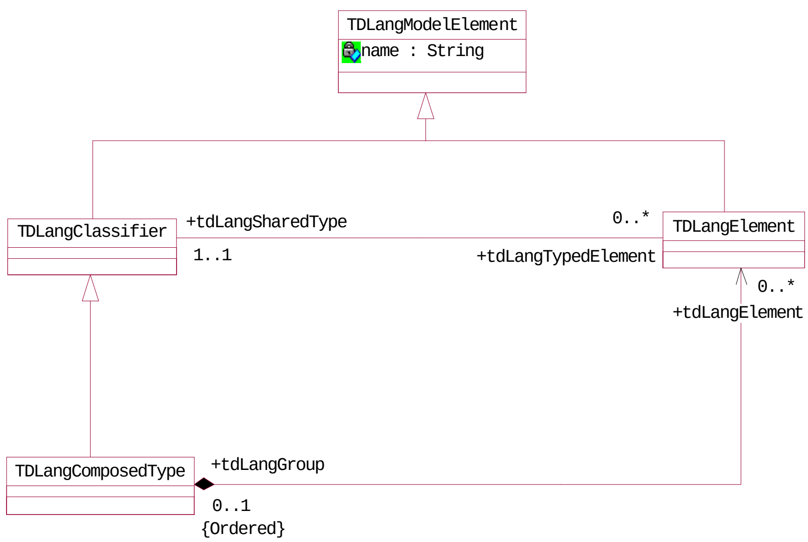



*Figure 7-4* TDLang Metamodel

## 7.3.4 TDLang Metamodel Descriptions

### 7.3.4.1 TDLangClassifier

TDLangClassifier is the parent class of all CAM language Classifier classes and TDLangComposedType. TDLangClassifier represents all data types of a CAM language metamodel. Since TDLangClassifier is abstract, it is implemented by language specific classifier classes. Sample subclasses of TDLangClassifier include String, integer, character, float, and addressable pointers for each language model. Subclasses of TDLangClassifier provide the type information declared by a TDLangElement.

### 7.3.4.2 tdLangTypedElement : TDLangElement

Used by the classifier associated to an element within a ComposedType to navigate back to the parent ComposedType.

### 7.3.4.3 TDLangComposedType

TDLangComposedType represents the type of data with subcomponents. TDLangComposedType is the parent class of all CAM language ComposedTypes. Since TDLangComposedType is abstract, it is implemented by language specific composed classes. Sample subclasses of TDLangComposedType are COBOL 01-level data declarations with nested elements, C structs and unions, and PL/I structures, unions, or elementary variables and arrays.

***tdLangElement : TDLangElement***

Used by TDLangComposedType to get a list of TDLangElements contained within the composed type.

### 7.3.4.4 TDLangElement

TDLangElement is the most basic, fundamental core class of the TDLang Metamodel. TDLangElement is the parent class of all CAM language element classes. TDLangElement represents typed unit elements declared in a copybook or source code, that is typed data elements without a subcomponent. Since TDLangElement is abstract, it is implemented by language specific element classes. Sample subclasses of TDLangElement are COBOLElement, CTypedElement, and PLIElement.

***tdLangGroup: TDLangComposedType***

Used by TDLangElement to determine the TDLangComposedType it belongs to.

***tdLangSharedType : TDLangClassifier***

Used by TDLangElement to determine the type associated to the element.

### 7.3.4.5 TDLangModelElement

TDLangModelElement is the parent class of all TDLang classes. Each instance of TDLangModelElement represents either a declared element or a classifier type. TDLangModelElements that represent a declared element can refer to another instance of TDLangModelElement that represents a classifier type. Since elements and user-defined types may have associated names, TDLangModelElement has a name attribute that can be separately instantiated by TDLangElement and TDLangClassifier.

### 7.3.5 Physical Representation Model: Type Descriptor Metamodel

Type Descriptor metamodel presents a language and platform independent way of describing implementation types, including arrays and structured types. This information is needed for marshaling and for connectors that have to transform data from one language and platform domain into another. Inspections of the type model for different languages can determine the conformance possibilities for the language types. For example, a **`long`** type in Java is often identical to a binary type (**`computational-5`**) in COBOL, and if so, the types may be inter-converted without side effect. On the other hand, an alphanumeric type in COBOL is fixed in size and if mapped to a Java type, loses this property. When converted back from Java to COBOL, the COBOL truncation rules may not apply, resulting in computation anomalies. In addition, tools that mix languages in a server environment (e.g., Java and COBOL in CICS and IMS) should find it useful as a way to determine how faithfully one language can represent the types of another. Therefore, an instance of the Type Descriptor metamodel describes the physical representation of a specific data type for a particular platform and compiler. The following figures illustrate the classes that constitute the Type Descriptor metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

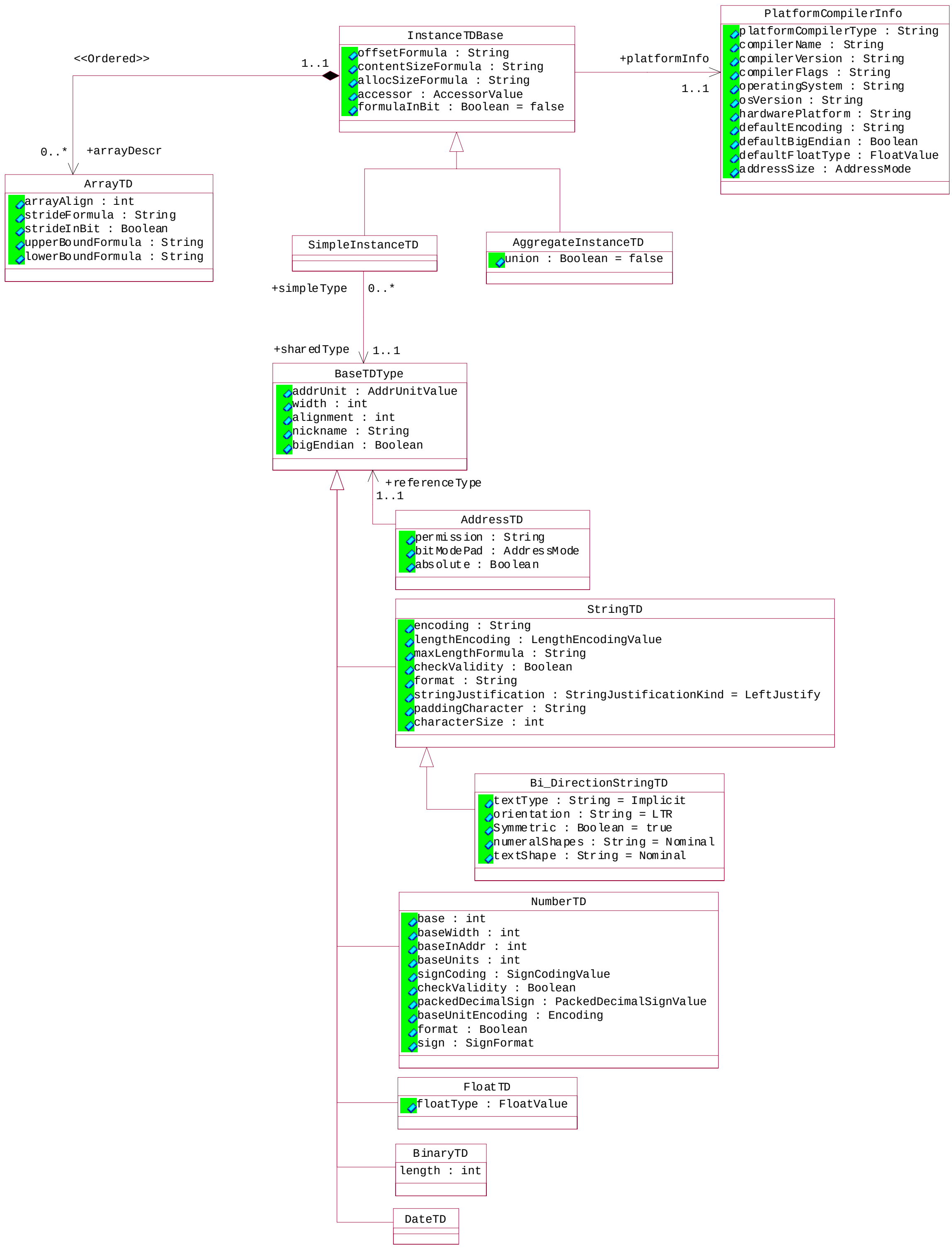



*Figure 7-5* Type Descriptor metamodel

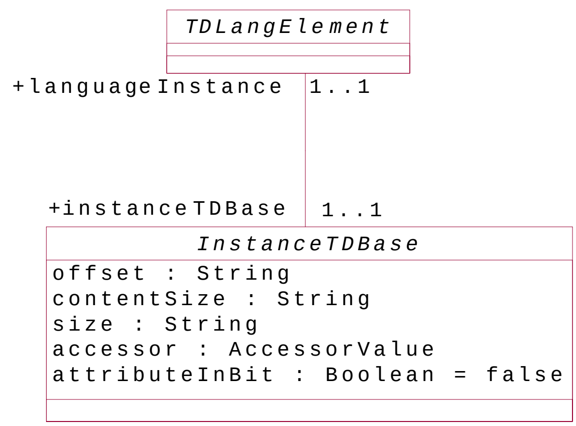



*Figure 7-6* TDLang to Type Descriptor


*Figure 7-7* Type Descriptor Enumerations

## 7.3.6 Type Descriptor Metamodel Descriptions

### 7.3.6.1 AddressTD

AddressTD represents pointers/addresses. Addresses should be considered to be different from NumberTD class because some languages on certain machines (e.g., IBM 400) represent addresses with additional information, such as permission type (which is not represented in NumberTD class).

### 7.3.6.2 ArrayTD

ArrayTD holds information for array types. Data element instances may be defined as repeating groups or arrays. This is modeled as a one-to-many association between InstanceTDBase and the ArrayTD model type. One instance of ArrayTD is created for each dimension, subscript, or independent index of the data element. Each instance holds information about the bounds and accessing computations. The association order between ArrayTD and InstanceTDBase is the same as the order for the corresponding association in the language model, and reflects the syntactic ordering of the indices as defined by the programming language.

### 7.3.6.3 BaseTDType

BaseTDType is the abstract parent class of all types in the TD Metamodel. BaseTDType holds implementation information common to all data types of the same runtime environment, as specified by PlatformCompilerInfo.

### 7.3.6.4 Bi-DirectionStringTD

Bi-DirectionStringTD is a set of optional attributes contained by PlatformCompilerInfo and StringTD. Bi-DirectionStringTD represents strings with extended properties and formats such as numeral shapes and right-to-left reading direction. When Bi-DirectionStringTD is contained by PlatformCompilerInfo, the bi-directional attributes will apply to the entire application program, whereas when Bi-DirectionString is contained by StringTD, the bi-directional attributes will only apply the specified string element.

### 7.3.6.5 BinaryTD

BinaryTD represents a string of binary bits whose format is not to be modified.

### 7.3.6.6 DateTD

DateTD represents date types with its associated format (e.g., mm/dd/yyyy, dd/mm/yyyy).

### 7.3.6.7 ExternalDecimalTD

ExternalDecimalTD represents numbers expressed in external decimal format.

### 7.3.6.8 FloatTD

FloatTD represents floating point numbers declared by a language element.

### 7.3.6.9 InstanceTDBase

InstanceTDBase is the most basic, fundamental core class of the Type Descriptor Metamodel. Every TD Metamodel instance contains at least one instance of InstanceTDBase. For each instance of a CAM language Element class there is a corresponding instance of InstanceTDBase. InstanceTDBase contains attributes that describe the physical layout of each declared variable and structure element in a program. It is an abstract class realized by either SimpleInstanceTD or AggregateInstanceTD. To find the parent of any instance (if it has one) navigate the association back to the CAM Language Element class (via a language-independent element class, e.g., TDLangElement), follow the association to the language-specific Composed class, then follow the association back to the parent InstanceTDBase.

### 7.3.6.10 IntegerTD

IntegerTD represents numbers expressed in binary format.

### 7.3.6.11 Number TD

NumberTD represents all integer and packed decimals. NumberTD is the parent class of ExternalDecimalTD, PackedDecimal, and IntegerTD.

### 7.3.6.12 PackedDecimalTD

PackedDecimalTD represents numbers expressed in packed decimal format.

### 7.3.6.13 PlatformCompilerInfo

PlatformCompilerInfo captures the static compiler and program runtime environment. Since this static information is shared by all instances of InstanceTDBase, this class only needs to be instantiated once.

### 7.3.6.14 SimpleInstanceTD and AggregateInstanceTD

Both SimpleInstanceTD and AggregateInstanceTD are subclasses of InstanceTDBase. InstanceTDBase has two concrete subtypes: SimpleInstanceTD and AggregateInstanceTD. SimpleInstanceTD models data elements without subcomponents, while AggregateInstanceTD models data elements with subcomponents. To find the subcomponents of an AggregateInstanceTD, one must navigate back to the corresponding data element declaration in the CAM language model. There, the association between an aggregate type and its subcomponents may be navigated, leading to a set of subcomponent data elements, each of which has one or more corresponding instances in the Type Descriptor model.

#### 7.3.6.15 StringTD

StringTD represents standard left-to-right format character strings. StringTD also supports single character elements.

#### 7.3.6.16 Type Descriptor Enumerations

- AccessorValue enumerates permission rights for each TDLangElement.
- AddressMode enumerates addressable units per hardware platform.
- AddrUnitValue enumerates the unit associated with the value of address attributes in Type Descriptor Metamodel.
- AlignType enumerates alignment delimiters for each data type specified by BaseTDType.
- ExternalDecimalSignValue enumerates various encoding methods to represent number characters.
- FloatValue enumerates floating types supported by Type Descriptor Metamodel.
- LengthEncodingValue enumerates string length encoding values supported by Type Descriptor Metamodel.
- NumeralShapes enumerates how Arabic numeric glyphic characters will be displayed on screen.
- Orientation enumerates how text should be presented from layout in memory.
- SignCodingValue enumerates numeric sign encoding values supported by Type Descriptor Metamodel.
- SignFormatValue enumerates the position of the positive and negative sign in a numeric item.
- StringJustificationKind enumerates string justification layout values supported by Type Descriptor Metamodel.
- TextShapes enumerates the shape of Arabic characters in relation to its position to neighboring characters.
- TypeOfText enumerates the method text should be read from memory. Text in memory can either be interpreted as it is logically implied by context or as it should be displayed visually on screen.

### 7.3.7 Type Descriptor Formulas

In the following discussion, “field” refers to a component of a language data structure described by the Type Descriptor metamodel, while “attribute” denotes part of the model, and has a value representing a “property” of the field. Thus the value of a field means a run-time value in a particular instance of a language data structure, whereas the value of an attribute is part of the description of a field in a language data structure, applies to all instances of the data structure, and is determined when the data structure is modeled.

For most attributes in an instance of the Type Descriptor metamodel, the value of the attribute is known when the instance is built, because the properties of the fields being described, such as size and offset within the data structure, are invariant. But if a field in a data structure is defined using the **`COBOL OCCURS DEPENDING ON`** construct or the PL/I **`Refer`** construct, then some properties of the field (and properties of other fields that depend on that field’s value) cannot be determined when the model instance is built.

Properties that can be defined using these language constructs are string lengths and array bounds. A property that could indirectly depend on these language constructs is the offset of a field within a structure, if the field follows a variable-size field.

To handle these language constructs, properties of a field that could depend on these constructs (and thus the values of the corresponding attributes) are defined with strings that specify a formula that can be evaluated when the model is used.

However, if a property of a field is known when the model instance is built, then the attribute formula simply specifies an integer value. For example, if a string has length 17, then the formula for its length is “17.”

The formulas mentioned above are limited to the following:

- Unsigned integers
- The following arithmetic integer functions

```
neg(x)   := -x         // prefix negate
add(x,y) := x+y        // infix add
sub(x,y) := x-y        // infix subtract
mpy(x,y) := x*y        // infix multiply
div(x,y) := x/y        // infix divide
max(x,y) := max(x,y)
min(x,y) := min(x,y)
mod(x,y) := x mod y
```

  The mod function is defined as mod(x,y) = r where r is the smallest non-negative integer such that x-r is evenly divisible by y. So mod(7,4) is 3, but mod(-7,4) is 1. If y is a power of 2, then mod(x,y) is equal to the bitwise-and of x and y-1.

- The val function

  The val function returns the value of a field described by the model. The val function takes one or more arguments, and the first argument refers to the level-1 data structure containing the field, and must be either:

- The name of a level-1 data structure in the language model.
- The integer 1, indicating the level-1 parent of the variable-size field. In this case, the variable-size field and the field that specifies its size are in the same data structure, and so have a common level-1 parent.

Here level-1 data structures refer to the top level declaration of a composed type element and a level-1 parent refers to 01 element that contains the field in question. An example of a level-1 data structure in COBOL would be an 01 element that contains other elements. The subsequent arguments are integers that specify the ordinal number within its substructure of the (sub)field that should be dereferenced.

By default, COBOL data fields within a structure are not aligned on type-specific boundaries in storage. For example, the "natural" alignment for a four-byte integer is a full-word storage boundary. Such alignment can be specified by using the **SYNCHRONIZED** clause on the declaration. Otherwise, data fields start immediately after the end of the preceding field in the structure. Since COBOL does not have bit data, fields always start on a whole byte boundary.

For PL/I, the situation is more complicated. Alignment is controlled by the Aligned and Unaligned declaration attributes. By contrast with COBOL, most types of data, notably binary or floating-point numbers, are aligned on their natural boundaries by default.

## 7.3.8 Type Descriptor Formula Examples

### 7.3.8.1 COBOL

The examples use the proposed inline comment indicator "*>" from the draft standard. It is not yet legal COBOL usage.

1. Consider the following data description:

```
*> Field                                   Offset

   01 Used-Car.                            *>     "0"
    02 Summary.                            *>     "0"
       03 Make pic x(36).                  *>     "0"
       03 Model pic x(44).                 *>     "36"
       03 VIN pic x(13).                   *>     "80"
       03 Color pic x(10).                 *>     "93"
         88 Red value 'Red'.
         88 White value 'White'.
         88 Blue value 'Blue'.
     02 History.                           *>     "103"
       03 Mileage pic 9(6).                *>     "103"
       03 NumClaims binary pic 9.          *>     "109"
       03 InsCode pic x.                   *>     "111"
       03 Claims.                          *>     "112"
         04 Claim occurs 1 to 9 times
             depending on NumClaims.       *> stride(1) = "157"
             05 ClaimNo pic x(14).         *>     "112"
             05 ClaimAmt binary pic 9(5).  *>     "126"
```

```
            05 Insurer pic x(39).        *>     "130"
            05 Details pic x(100).       *>     "169"
      02 Price comp pic 9(5)v99.         *>
"add(112,mpy(val(1,2,2),157))"
```

The offset of **Model** is straightforward, and is given by the formula "**36**." So is that of **Claims**, which is "**112**."

But because the array **Claim** can occur a variable number of times, the structure **History** is a variable-size field. Thus the offset of **Price**, which immediately follows **Claims**, requires a more complicated formula, involving the array stride (the distance between successive elements along a specific dimension). In the case when there is only one dimension for **Claim**, the formula for its stride is "**157**." Thus the formula offset of **Price** for a single dimension **Claim** is:

**"add(112,mpy(val(1,2,2),157))"**

The first argument of the val function is 1, meaning that the field containing the value at run-time, **NumClaims**, is in the same level-1 structure, **Used-Car**, as the field, **Price**, whose offset is specified by the formula. The other two arguments are 2 and 2. The first 2 refers to the second immediate subcomponent, **History**, of **Used-Car**. The second 2 means that the field to be dereferenced is the second component of **History**, that is, **NumClaims**.

In the case when **NumClaims** is greater than 1 (i.e., when **Claims** is a multi-dimension array) the offset for each element within **Claims** is **157** more than the offset for the previous dimension. For example, the offset formula for the second instance of **ClaimNo** is 112+157=269 while the third instance would be 269+157=426.

If the **OCCURS DEPENDING ON** object were in a separate structure, the third subcomponent of level-1 structure **Car-Data**, say, then the val function would be "**val(Car-Data,3)**."

2. COBOL structure mapping is top-down, although the direction doesn't make any difference unless the SYNCHRONIZED clause is specified on the data declaration. Specifying SYNCHRONIZED forces alignment of individual fields on their natural boundaries, and thus introduces "gaps" into the structure mapping. Consider the following data structure that is identical to the previous example, except for the SYNCHRONIZED clause:

```
*> Field                                   Offset

   01 Used-Car sync.                       *> "0"
     02 Summary.                           *> "0"
       03 Make pic x(36).                  *> "0"
       03 Model pic x(44).                 *> "36"
       03 VIN pic x(13).                   *> "80"
       03 Color pic x(10).                 *> "93"
         88 Red value 'Red'.
         88 White value 'White'.
         88 Blue value 'Blue'.
     02 History.                           *> "103"
```

```
          03 Mileage pic 9(6).                *> "103"
          03 NumClaims binary pic 9.          *> "110"
          03 InsCode pic x.                   *> "112"
          03 Claims.                          *> "113"
            04 Claim occurs 1 to 9 times
                depending on NumClaims.       *> stride(1) = "160"
                       05 ClaimNo pic x(14).  *> "113" plus one
                                              slack byte after each
                                              instance of ClaimNo
                       05 ClaimAmt binary pic 9(5).  *> "128"
                       05 Insurer pic x(39).         *> "132"
                       05 Details pic x(100).        *> "171" plus
                                              one slack byte after
                                              each instance of
                                              Details and one slack
                                              byte after each
                                              instance of Claims
         02 Price comp pic 9(5)v99.        *>
"add(add(113,mpy(val(1,2,2),160)),3)"
```

To position the binary fields on their appropriate half-word or full-word storage boundaries, COBOL introduces padding, known as “slack bytes,” into the structure. Working top-down, this padding is introduced immediately before the field needing alignment. So there is one byte of padding between **`Mileage`** and **`NumClaims`**.

For an array, such as Claim, COBOL not only adjusts the padding within an element, but also the alignment of each element of the array. In the example, the first occurrence of Claim starts one byte past a full-word boundary. Because the field ClaimNo is three and a half words long, it ends three bytes past a full-word boundary, so COBOL inserts one byte of padding immediately before the binary full-word integer ClaimAmt. And to align subsequent occurrences, so that they too start one byte past a full-word boundary like the first, and can thus have an identical configuration, COBOL adds two bytes of padding at the end of each occurrence.

Finally, after padding, each occurrence of Claim (starts and) ends one byte past a full-word boundary, so COBOL puts three bytes of padding before the binary field Price. As a result of all these extra bytes, the formula for the offset of Price has changed considerably from the unaligned example, and is now:

**`"add(add(113,mpy(val(1,2,2),160)),3)"`**

There are several differences between the **`OCCURS DEPENDING ON`** construct and PL/I’s Refer option. Storage for COBOL structures is always allocated at the maximum size, whereas PL/I structures are allocated at the actual size specified by the Refer option. It is legal and usual to change the number of occurrences in a particular instance of a variable-size COBOL array, and this has the effect of changing the location and offset of any fields that follow the array. For PL/I, the value of the Refer object of a particular instance of a structure is intended to be fixed during execution. Thus aligned objects following a variable-size field are always correctly aligned for each instance of the structure, because the amount of padding is computed uniquely for each instance, as determined by the Refer option. By contrast, the amount of padding for any aligned fields following a variable-size COBOL array is computed assuming the maximum array size, and is fixed at compile time. If the array is smaller than its maximum size, then the alignment will typically be incorrect. For instance in this example:

```
1 a sync.
       2 b binary pic 9.
       2 c pic x occurs 1 to 5 times depending on b.
       2 d binary pic 9(9).
```

COBOL inserts one byte between `c` and `d`. The alignment of `d` is therefore correct for only two values of `b`, the maximum, 5, and 2.

3. As noted above, the formulas describe not only offsets of fields within a structure, but also properties of arrays, such as bounds and strides. COBOL does not have true multi-dimensional arrays, although element references do use multiple subscripts. Instead, COBOL has arrays of arrays, as in the following simple example:

```
1 a. *< offset = "0"
   2 d1 occurs 5 times.      *< offset = "0"
                             *< lbound(1) = "1"
                             *< hbound(1) = "5"
                             *< stride(1) = "168"
       3 d2 occurs 6 times. *< offset = "0"
                             *< lbound(2) = "1"
                             *< hbound(2) = "6"
                             *< stride(2) = "28"
          4 el binary pic 9(9) occurs 7 times.  *< offset = "0"
                             *< lbound(3) = "1"
                             *< hbound(3) = "7"
                       *< stride(3) = "4"
```

The program can refer to slices of the array by subscripting the higher-level container fields, for example, **`d1(2)`** or **`d2(3, 4)`**, but the normal kind of reference is to the low-level elements using the full sequence of subscripts, for instance, **`el(4, 5, 6)`**. To locate element **`el(m, n, o)`** using these stride formulas, one would take the address of **a** and add to it (m-1)*168 + (n-1)*28 + (o-1)*4. For COBOL, the lower bound of an array subscript is always 1. That is, the first element is always element(1), and vice versa.

Needless to say, any dimension of the array can have the **`OCCURS DEPENDING ON`** clause, and the array can be followed by other fields that complicates the formulas a lot. Consider the example:

```
1 a.
   2 x1 binary pic 9.           *< offset = "0"
   2 x2 binary pic 9.           *< offset = "2"
   2 x3 binary pic 9.           *< offset = "4"
   2 d1 occurs 1 to 5 times     *< offset = "6"
        depending on x1.        *< lbound(1) = "1"
                                *< hbound(1) = "val(1,1)"
                                *< stride(1) =
"mpy(val(1,2),mpy(val(1,3),4))"
       3 d2 occurs 1 to 6 times   *< offset = "6"
```

```
            depending on x2.    *< lbound(2) = "1"
                                *< hbound(2) = "val(1,2)"
                                *< stride(2) = "mpy(val(1,3),4)"
            4 el binary pic 9(9)      *< offset = "6"
               occurs 1 to 7 times    *< lbound(3) = "1"
            depending on x3.          *< hbound(3) = "val(1,3)"
                                      *< stride(3) = "4"
      2 b binary pic 9(5).            *< offset = "see below!"
```

Computing the address of a particular element still involves the stride formulas, but these are no longer simple integers. The address of element **el(m, n, o)** in the above example is given by taking the address of **a** and adding to it:

**(m-1)*stride(1) + (n-1)*stride(2) + (o-1)*stride(3), i.e.,**

**(m-1)*4*val(1,3)*val(1,2) + (n-1)*4*val(1,3) + (o-1)*4.**

Similarly, these stride formulas are used in the formula for the offset of b:

**"add(6,mpy(val(1,1),mpy(val(1,2),mpy(4,val(1,3)))))**

### 7.3.8.2 PL/I

1. Given the following structure:

```
dcl /* offset                                      */
         1 c unaligned              /*       "0"   */
           ,2 c1                    /*       "0"   */
             ,3 c2  fixed bin(31)   /*       "0"   */
             ,3 c3  fixed bin(31)   /*       "4"   */
           ,2 c4                    /*       "8"   */
             ,3 c5  fixed bin(31)   /*       "0"   */
             ,3 c6  fixed bin(31)   /*       "4"   */
             ,3 c7  fixed bin(31)   /*       "8"   */
           ,2 c8    fixed bin(31)   /*       "20"  */
           ,2 c9    char( * refer(c7) ) /*   "24"  */
           ,2 c10   char(6) /*   "add(24,val(1,2,3))" */
          ,2 c11    char(4) /*   "add(add(24,val(1,2,3)),6)" */
                              ;
```

The offset of c3 would be given by the simple formula “4” but the offset of c10 would be given by the formula:

**"add(24,val(1,2,3))"**

The first argument in the above val function is 1 that indicates the current structure, c. The subsequent arguments are 2 and 3, indicating that the third element, c7, of the second level-2 field, c4, is the field to be dereferenced.

The offset of c11 is equal to the offset of c10 plus the length of c10 and would be given by the following formula:

**"add(add(24,val(1,2,3)),6)"**

2. PL/I structure mapping is not top-down, and this can be illustrated by examining the mapping of the following structure:

```
dcl /* offset                              */
   1 a based,                        /* "0" */
      2 b,                           /* "0" */
       3 b1 fixed bin(15),    /*     "0"*/
       3 b2 fixed bin(15),    /*     "2"*/
       3 b3 fixed bin(31),    /*     "4"*/
      2 c,               /*   "add(8,mod(neg(val(1,1,1)),4))"*/
       3 c1 char( n refer(b1)),  /*     "0" */
       3 c2 fixed bin(31);       /*     "val(1,1,1)" */
```

The value of b1 is given by val(1,1,1), and in order to put c2 on a 4-byte boundary, PL/I puts any needed padding before c (yes, not between c1 and c2), and hence the offset of c would be given by the following formula:

```
"add(8,mod(neg(val(1,1,1)),4))"
```

So if b1 contains the value 3, then this formula becomes add(8,mod(neg(3),4)), which evaluates to 9 (i.e., there is one byte of padding between the structure b and the structure c).

3. The model also uses these formulas to specify the bounds and strides in an array, where the stride is defined as the distance between two successive elements in an array.

   For example, in the following structure, the second dimension of a.e has a stride specified by the formula “4” and the first dimension by the formula “20”:

```
dcl
   1 a,                             /* offset    = "0" */
      2 b(4) fixed bin(31),         /* offset    = "0" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "4" */
      2 c(4) fixed bin(31),         /* offset    = "16" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "4" */
      2 d(4) char(7) varying,       /* offset    = "32" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "9" */
      2 e(4,5) fixed bin(31);       /* offset    = "68" */
                                    /* lbound(1) = "1" */
                                    /* hbound(1) = "4" */
                                    /* stride(1) = "20" */
                                    /* lbound(2) = "1" */
                                    /* hbound(2) = "5" */
                                    /* stride(1) = "4" */
```

   This means that to locate the element a.e(m,n), one would take the address of a.e and add to it (m-1)*20 + (n-1)*4.

If the example were changed slightly to:

```
dcl
   1 a(4),                      /* offset = "0"       */
                                /* lbound(1) = "1"    */
                                /* hbound(1) = "4"    */
                                /* stride(1) = "40"   */
      2 b fixed bin(31),        /* offset = "0"       */
      2 c fixed bin(31),        /* offset = "4"       */
      2 d char(7) varying,      /* offset = "8"       */
      2 e(5) fixed bin(31);     /* offset = "20"      */
                                /* lbound(1) = "1"    */
                                /* hbound(1) = "5"    */
                                /* stride(1) = "4"    */
```

then there is padding between d and e, but the user of the type descriptor can be blissfully unaware and simply use the stride and offset formulas to locate any given array element.

The stride for a is “40,” the stride for e is “4,” and the offset for e is “20.” This means that to locate the element a(m).e(n), one would take the address of a and add to it (m-1)*40 + 20 + (n-1)*4.

Finally, if the example were changed again to:

```
dcl
   1 a(4),                     /* offset = "0"          */
                               /* lbound(1) = "1"       */
                               /* hbound(1) = "4"       */
                               /* stride(1) = "40"      */
      2 b fixed bin(31),       /* offset = "0"          */
      2 c(8) bit(4),           /* offset = "4"          */
                               /* lbound(1) = "1"       */
                               /* hbound(1) = "8"       */
                               /* stride(1) = "4"       */
      2 d char(7) varying,     /* offset = "8"          */
      2 e(5) fixed bin(31);    /* offset = "20"         */
                               /* lbound(1) = "1"       */
                               /* hbound(1) = "5"       */
                               /* stride(1) = "4"       */
```

then the computations for a.e are the same as above, but the computations for a.c become interesting.

The stride for a is still “40,” the stride for c is “4” (but this “4” is a count of bits, not bytes), and the byte offset for c is “4.” To locate the element a(m).c(n), one needs both a byte address and a bit offset. For the byte address, one would take the address of a and add to it (m-1)*40 + 4 + ((n-1)*4)/8. The bit offset of a(m).c(n) would be given by mod((n-1)*4,8).

## 7.3.9 Physical Representation Model: TDLang Interaction Diagram

The purpose of this section is to provide the reader with an overview of how the TDLang model is used to connect and integrate CAM, Type Descriptor, and interface models together. The Type Descriptor metamodel is a language-independent model used to convert a datatype into its expected language-specific type. This is accomplished by associating the base class, InstanceTDBase, to TDLangElement. As the parent class of all language model element classes, TDLangElement allows Type Descriptor to access the information regarding all language-specific data types for marshaling. Type Descriptor's association to the language elements via TDLangElement also provides the aggregate associations captured in the language models (i.e., the ComposedTypes associations for parent-child relationships). This ability to navigate up to parent or sibling elements is required to determine the value of various formula-based attributes in the Type Descriptor model. For example, in order for a child element C to determine its offset formula value, it will need to navigate up to element B to find B's offset value and allocation size. The result of the adding element B's offset value and allocation size is element C's offset value.

Caching and navigation are two approaches to determining the parent value, but the navigation approach is superior to the cache approach in two respects. First, contents in the cache may become invalid as subscript values change from one child element to the next during runtime, resulting inaccurate cache data. Second, to fix this problem the marshaller will need to recalculate the values of each element at runtime, resulting in a decrease in performance. In the case when we apply navigation from the Type Descriptor model to the language models, we are able to quickly go from the child to the parent element to determine the formula information on a real-time basis. The navigation approach provides accurate values quickly without the need to perform recalculations.

The next diagram shows how language models associate to the Type Descriptor model via the TDLang model. Following the diagram is a brief explanation of what each class represents.

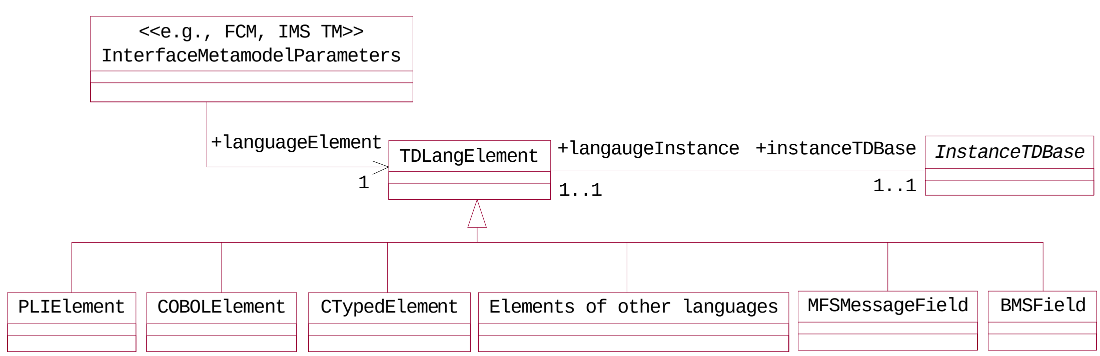



*Figure 7-8* A View of Select CAM Models and Interface Models interacting with TDLang

## 7.3.10 Descriptions of TDLang Interaction Diagram

### 7.3.10.1 Interface Metamodel Parameters

Interface Metamodel Parameters represent a variety of input and output parameter classes that map to underlying language elements. Information on the language element's physical representation is captured by the Type Descriptor metamodel. Each instance of TDLangElement maps its corresponding physical representation in InstanceTDBase. TDLangElement navigates to InstanceTDBase via the instanceTDBase association. Examples of Enterprise Application Metamodel Parameters include ApplicationData (from IMS Transaction Message Metamodel), MFSMessageField (from IMS MFS Metamodel), and FCMParameter (from FCM Metamodel).

### 7.3.10.2 TDLangElement and Language Elements

As stated in Section 7.3.4.4, "TDLangElement," on page 7-8, TDLangElement is the parent class of all CAM language Element classes. Figure 7-8 on page 7-23 shows how any CAM language element can be modeled to support any given Interface Metamodel Parameter.

### 7.3.10.3 InstanceTDBase

As stated in Section 7.3.6.9, "InstanceTDBase," on page 7-13. InstanceTDBase is used to represent the physical layout of each language element.

## 7.3.11 Sample Serialization of Convergent Metamodel

An example of how a marshaller might traverse the Type Descriptor-TDLang-Language model is as follows:

Given the following COBOL Data Declaration:

```
01     NAME.
     02     FIRST     PIC X(10).
     02     LAST      PIC X(10).
```

The following COBOL and Type Descriptor XMI instances would be serialized:


```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:COBOL="COBOL.xmi" xmlns:TypeDescriptor="TypeDescriptor.xmi">
  <COBOL:COBOLElement xmi:id="Element:NAME" name="NAME" level="01"
instanceTDBase="AggregateInstanceTD_1" sharedType="Type:NAME"/>
  <COBOL:COBOLComposedType xmi:id="Type:NAME">
    <element xmi:id="Element:NAME/FIRST" name="FIRST" level="02"
redefined="false" instanceTDBase="SimpleInstanceTD_1"
sharedType="Type:NAME/FIRST" initial="COBOLElementInitialValue_1"/>
    <element xmi:id="Element:NAME/LAST" name="LAST" level="02"
redefined="false" instanceTDBase="SimpleInstanceTD_2"
sharedType="Type:NAME/LAST" initial="COBOLElementInitialValue_2"/>
  </COBOL:COBOLComposedType>
  <TypeDescriptor:PlatformCompilerInfo xmi:id="PlatformCompilerInfo_1"
language="COBOL" defaultCodepage="8859_1" defaultBigEndian="false"
defaultFloatType="ieeeNonExtended" defaultExternalDecimalSign="ascii"
defaultAddressSize="mode32"/>
  <TypeDescriptor:AggregateInstanceTD xmi:id="AggregateInstanceTD_1"
offset="0" contentSize="20" size="20" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME"/>
  <COBOL:COBOLElementInitialValue xmi:id="COBOLElementInitialValue_1"
name="FIRST" initVal=" " valueKind="all_literal"/>
  <COBOL:COBOLAlphaNumericType xmi:id="Type:NAME/FIRST" usage="display"
pictureString="XXXXXXXXXX" synchronized="false" justifyRight="false"/>
  <TypeDescriptor:SimpleInstanceTD xmi:id="SimpleInstanceTD_1"
offset="0" contentSize="10" size="10" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME/FIRST" sharedType="StringTD_1"/>
  <TypeDescriptor:StringTD xmi:id="StringTD_1" addrUnit="word"
width="10" alignment="byte" lengthEncoding="fixedLength"
prefixLength="0" stringJustification="leftJustify" paddingCharacter=" "
characterSize="1"/>
  <COBOL:COBOLElementInitialValue xmi:id="COBOLElementInitialValue_2"
name="LAST" initVal=" " valueKind="all_literal"/>
  <COBOL:COBOLAlphaNumericType xmi:id="Type:NAME/LAST" usage="display"
pictureString="XXXXXXXXXX" synchronized="false" justifyRight="false"/>
  <TypeDescriptor:SimpleInstanceTD xmi:id="SimpleInstanceTD_2"
offset="10" contentSize="10" size="10" accessor="readWrite"
attributeInBit="false" platformInfo="PlatformCompilerInfo_1"
languageInstance="Element:NAME/LAST" sharedType="StringTD_2"/>
  <TypeDescriptor:StringTD xmi:id="StringTD_2" addrUnit="word"
width="10" alignment="byte" lengthEncoding="fixedLength"
prefixLength="0" stringJustification="leftJustify" paddingCharacter=" "
characterSize="1"/>
</xmi:XMI>
```


Of particular interest is how the offsetFormula is determined. To determine the offsetFormula value of element LAST, the model needs to be able to navigate upward from LAST's SimpleInstanceTD to FIRST's SimpleInstanceTD to determine the offsetFormula and allocSizeFormula attributes of FIRST. Formula-based values can either be static (serialized during import time) or dynamic (serialized during runtime). It is this capability to navigate back-and-forth from language models to Type Descriptor that allows us to determine how to marshal each language element.

Formula-based attributes in the Type Descriptor model are typed as String in order to support both calculation and numeric values. Runtime determined values such as COBOL's Occurs-Depending-On clause will have calculation formulas as its value (e.g., "20+10x") while static values will use numeric values (e.g., allocSizeFormula of FIRST is "10"). Calculation formulas will be evaluated by a "Formula Evaluator," which takes the formula String as input and returns the calculated numeric value when runtime information is available (e.g., once the 'x' value of formula "20+10x" is determined we can return a numeric value). In the case of a numeric value (evaluated integer), simply pass the attribute value into a "Formula Evaluator" program and the integer representation of the string will be returned. The formulas in the Type Descriptor model should be generic for all languages. Therefore, the "Formula Evaluator" will cover all languages (COBOL, C, C++, PL/I, etc.).

# *Part 3 - Profile Definition*

## *Contents*

This section contains the following chapters.



The profile presented here focuses on two main modeling approaches, based on collaborations and based on activities. These are described in Chapters 8 and 9, respectively.

The collaboration-modeling approach is based on a modeling framework of classes that provide detailed definitions of the semantics of the collaboration. It is thus useful for providing the detailed specification of message flows in the design of integration subsystems.

The activity-modeling approach is based on the use of activity graphs. This approach is particularly useful for showing the overall control and data flow required for integration, typically at a higher level than in collaboration modeling.

Casting the metamodel as a UML profile allows EAI architecture models to be notated using standard UML notation. This means that most UML tools (specifically ones which support the extension mechanisms of UML, such as stereotypes and tagged values) can be used to define EAI architecture models.

Standard practice for defining UML profiles has been adopted. A mapping of metamodel classes to their base UML classes, with accompanying stereotypes, tagged values and constraints is summarised for each approach. An implementation of this mapping can be used, for example, to generate metadata conforming to the EAI metamodel from XMI generated from models notated using the UML profile. Specialized EAI tools will more likely use the metamodel than the UML profile as a basis for storing and manipulating models.

The art of defining a UML profile is to provide the best fit possible with UML, so that the notation is natural for a modeler in the relevant domain (EAI in this case), and fits with one's general intuitions about the the meaning of the elements of UML that are used in the profile. The profile described here has been designed with these principles in mind.

# Collaboration Modeling 8

## 8.1 Overview

### 8.1.1 General Approach

The Collaboration Profile for EAI makes use of UML class and collaboration diagrams to notate EAI models. The main parts of the profile are:

- Notation for terminals
- Notation for operators
- Notation for resources
- Notation for message formats

Operators are notated by class diagrams, which declare the input and output terminals of the operator and the message formats of those terminals. The class diagram can also be annotated with the definition of the operations performed when manipulating incoming messages to generate outgoing messages.

For compound operators, class diagrams also specify the component operators of the compound, which may, themselves, be compound operators. Collaboration diagrams are used to show how its components are connected together.

Different kinds of terminals are defined by appropriate stereotypes on UML Class. Specific, named terminals are identified with operators via associations.

Different kinds of operator are identified by appropriate stereotypes on UML Class.

Some operators make use of resources. Resources are notated by classes, with stereotypes used to capture the different kinds of format.

Message formats are notated by classes, with stereotypes used to capture the different kinds of format.

### 8.1.2 Use of UML Operations

There are places where UML operations have been used with specific names to 'carry' certain pieces of metadata within a model defined by the profile. For example, when one defines a terminal, it is necessary to define an operation called handle whose return type determines the format of message content that the terminal can handle; when one defines a filter, it is necessary to define a boolean operation allow that determines, for a message supplied as argument, the conditions under which a message can pass through the filter. This approach to encoding this information was taken, because it accords with one's intuitions about the meaning of UML and of UML operations in particular. For example, one is able to explain what a filter does by referring to its allow operation - only incoming messages for which the allow operation evaluates to true get passed on.

It should be stressed that the operations themselves imply nothing about the scheme used to implement models, though clearly the information they hold will need to be carried through in some way. Indeed, most implementations are likely to work from the metamodel direct (as this issue does not arise there) and the profile just used as a means of defining models using UML notation, which can then get converted to instances of the metamodel for subsequent processing.

Any definition of an operation used in operator specifications must be provided as part of the specification of that operation. There are many ways to show the definition of UML operations, which will depend on specific organizational practices and/or support provided by UML CASE tools. If tools do not support the display of operation specifications on diagrams (as many don't) a UML note may be used in addition to repeat the definition on the diagram. In this document, the specification of operations in examples is relayed by notes on the diagrams.

### 8.1.3 Concrete Notation

Only raw stereotypes have been defined in this profile. The user may replace these with concrete icons at his or her discretion.

### 8.1.4 Chapter Structure

The remainder of this chapter provides a detailed description of each of the four parts of the profile. Each part is described stereotype by stereotype, using generic examples for illustration. The constraints that apply in the context of a particular stereotype are also defined. The detailed descriptions are followed by a section describing the mapping of the EAI metamodel to the elements of the profile. This section also provides a summary of the stereotypes used in the profile, and follows the format laid down by UML 1.4.

## 8.2 Terminals

The terminals of an operator are shown by associations to classes with stereotypes <<input>> (for input terminals) and <<output>> (for output terminals), from classes with operator stereotypes (see sections below). Figure 8-1 gives a prototypical example, showing the definition of terminals for a primitive operator. As shown, the primitive operator has two input terminals, names “in” and “queueIn.” While both these terminals handle the same kind of message format, the latter is specifically known to be a queued terminal. The primitive operator is also shown to have two output terminals, named “out1” and “out2.”

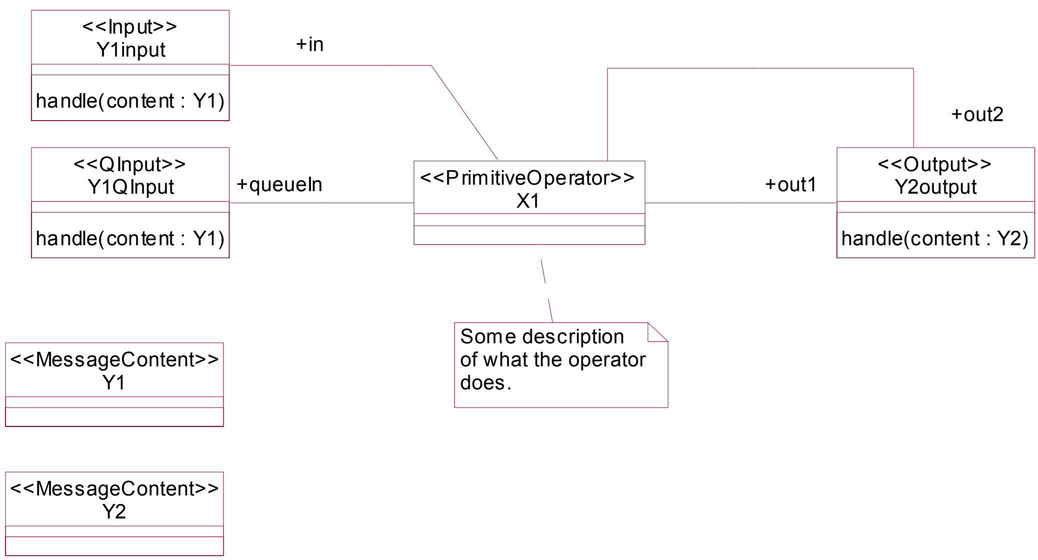



*Figure 8-1* Class diagram for prototypical primitive operator with terminals

An input terminal is responsible for conveying incoming messages to the operator, while an output terminal is responsible for conveying outgoing messages away from the operator. The names of the terminals with respect to the operator are specified as labels on the appropriate association end. The associations are navigable only from operator to terminal, and they have cardinality 1. These markings (which may never change) may be omitted from the diagram (tool permitting) to avoid clutter. Any other properties and inconsistent with the profile. In general, operators may have one or more input and one or more output terminals. The number and names of the input and output terminals may be constrained for specialist primitive and compound operators.

Terminals can handle messages with a specified content format. This is indicated by declaring an operation handle on the class defining terminal kinds (i.e., classes with stereotypes <<input>> and <<output>>) that takes one argument of the specified format. Formats are specified by classes with a stereotype <<LangElement>> or one of its substereotypes, or stereotype <<MessageContent>> or one of its substereotypes. For most operators (adapters are the exceptions), the stereotype will usually be <<MessageContent>> corresponding to the generic format for message content.

It is not the role of this specification to say how a terminal handles its messages. However, the stereotypes <<QInput>> and <<QOutput>> may be used to indicate that handling is performed using a queue. Unless stated otherwise (e.g., as a constraint), it is assumed that terminals defined for any kind of operator may be plain or queued.

Finally, dynamic connection of terminals is supported. That is, it is possible to send some operators (for example routers) a message containing a terminal identifier, so that the operator can add or remove that terminal from the list of targets of one or more of its output terminals. The targets of an output terminal are the terminals connected to it.

***Constraints***

There should only be one input and output class per handle format/stereotype pairing, and the name of this class will be a concatenation of the format name and the stereotype name.

The type of the content parameter of the handle operation must have a stereotype of <<LangElement>> or one of its sub-stereotypes, or of <<MessageContent>> or one of its substereotypes.

## 8.3 Operators

### 8.3.1 Primitive Operator

Figure 8-1 on page 8-3 also shows a prototypical example of the definition of a primitive operator.

Primitive operators are useful for notating operators that have no internal structure (or whose internal structure is of no interest) such as system applications. A generic primitive operator is shown as a class with a stereotype <<PrimitiveOperator>>. The class may have an associated note (corresponding to EAIAnnotation in the metamodel) for recording a description of what the operator does.

***Constraints***

The type of content of the terminals of a generic primitive operator must have a stereotype <<MessageContent>> or one of its substereotypes.

### 8.3.2 Transformers and Database Transformers

Figure 8-2 shows the general format of the notation used to define a transformer, which is represented by a class with stereotype <<Transformer>>. A transformer uses the transform operation to transform the content of the input message and then sends the transformed message via the single output terminal of the transformer.

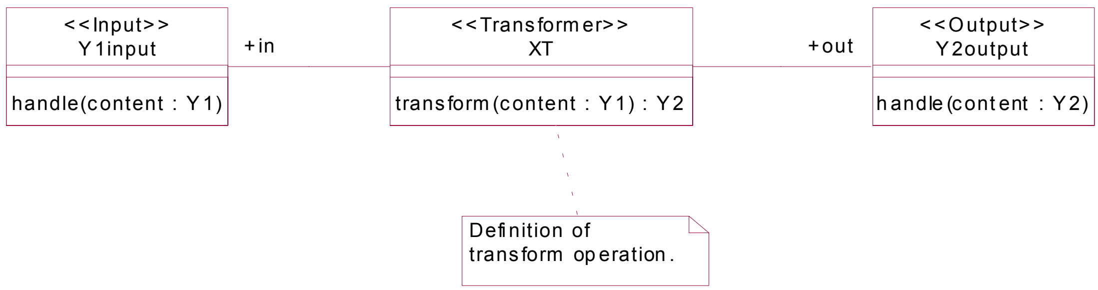



*Figure 8-2* Class diagram for prototypical transformer

A database transformer is just like a transformer, except that it accesses a database in order to perform the transform operation. In this case, the stereotype <<DBTransformer>> is used, and this requires a database resource to be declared, as in Figure 8-3.

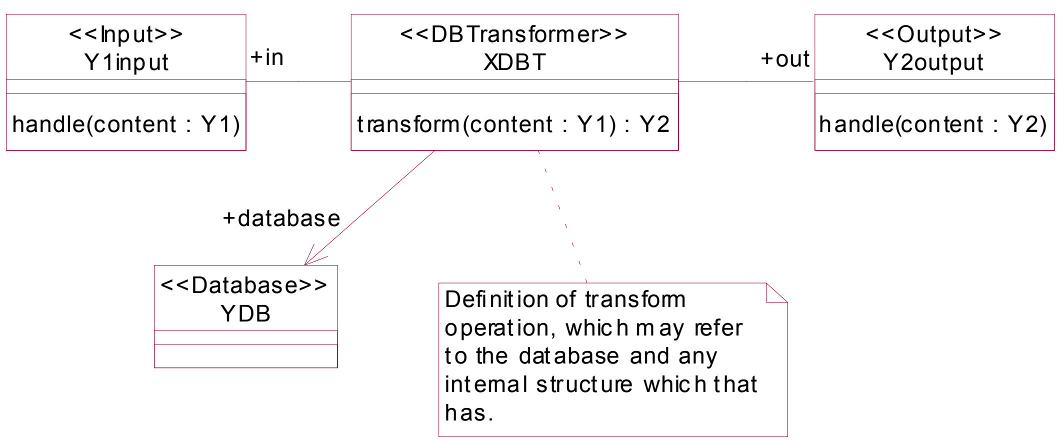



*Figure 8-3* Class diagram for prototypical database transformers

Additionally, the definition of transform may make reference to this attribute.

### *Constraints*

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the transform operation.

The type of content of the terminals of a transformer must have a stereotype <<MessageContent>> or one of its substereotypes.

For database transformers, there must be a directed association to a database resource (i.e., a class with stereotype <<Database>>) with the rolename “database” at the database resource end.

## 8.3.3 Filters

Figure 8-4 shows the general format of the notation used to define a filter.

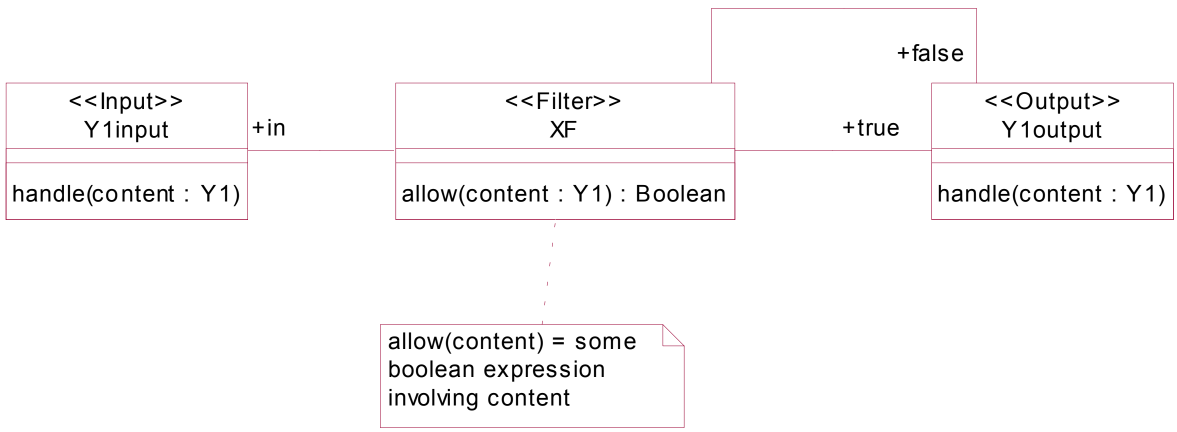



*Figure 8-4* Class diagram for prototypical filter

A filter does not modify the content of the messages it receives. However, a filter only passes on those messages whose content meets specific criteria. When a filter is triggered, it uses the allow operation to test if the content of the input message meets the criteria. If so, the content is sent to the true output terminal, otherwise it is sent to the false terminal.

***Constraints***

The content format of the input and output terminals must match that of the parameter of the allow operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

## 8.3.4 Streams

For operators described so far it is assumed that messages are always received in the order that they are sent and that there is basically no delay in their transmission. In reality, there are some cases where a stream of messages may be received in a different order than that in which they are sent and they may be received at a different rate than that at which they are sent. A stream operator is used to model this. Figure 8-5 shows the general format of the notation used to define a stream operator.

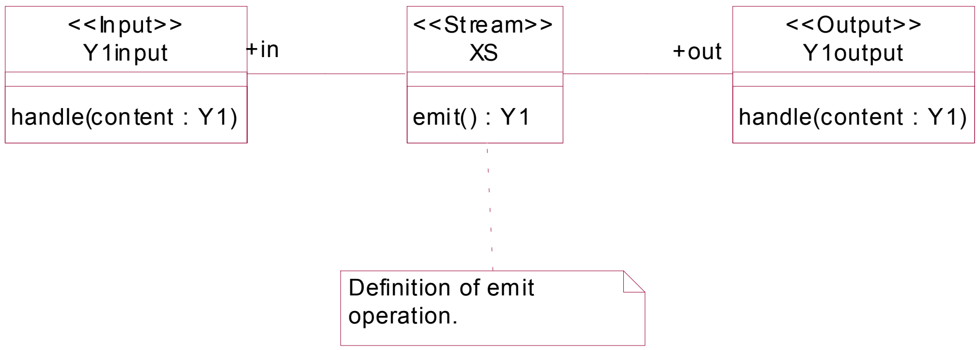



*Figure 8-5* Class diagram for prototypical stream

Messages that arrive from the input terminal do not get passed on, but instead are stored in a buffer or some other appropriate data structure. The emit operation defines the algorithm used to decide when and in what order messages get emitted to the output terminal. Abstractly, one can imagine a loop that continually calls the emit operation. It returns a message to be put on the output terminal at each call. There may be a delay between its being called and its returning a message.

### *Constraints*

The content format of the terminals must match that of the result of the emit operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

## *8.3.5 Post Daters*

Figure 8-6 shows the general format of the notation used to define a post dater.

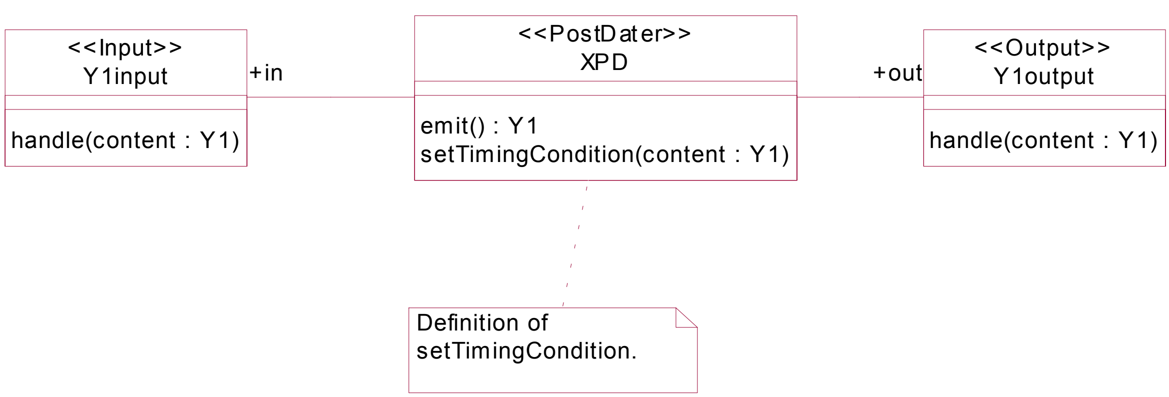



*Figure 8-6* Class diagram for prototypical post dater

A post dater is specified using the <<PostDater>> stereotype. A special kind of stream is a post dater. On receipt of a message at its input terminal, it adds the message to the buffer, and creates an individual timingCondition for it. The timingCondition is derived from the content of the input message by the setTimingCondition operation. A post dater holds the message until its individual timing condition is met and then emits it from its out terminal.

As the definition for emit is fixed for post daters, only a definition for setTimingCondition should be provided.

### *Constraints*

The content format of the terminals must match that of the result of the emit operation and the parameter of the setTimingCondition operation. This type must have a stereotype <<MessageContent>> or one of its substereotypes.

## *8.3.6 Aggregators*

Figure 8-7 shows the general format of the notation used to define an aggregator.

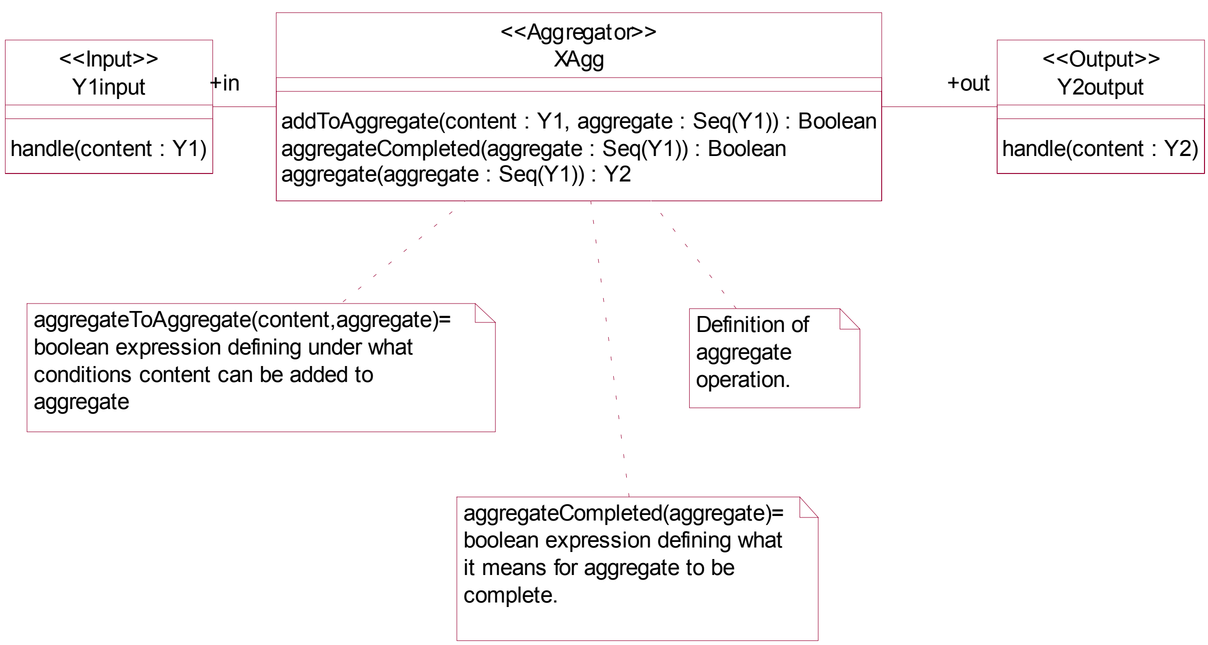



*Figure 8-7* Class diagram for prototypical aggregator

An aggregator operator is indicated by the <<Aggregator>> stereotype. It creates aggregate messages based on one or more message aggregation specification, each of which is modeled by an associated class with the <<MessageAggregation>> stereotype. (Note that an aggregator can create multiple aggregates either by having an association with a multiplicity of greater than one with the same message aggregation class, in which case all aggregates share the same specification, or by having multiple associations with different message aggregation classes.)

On receipt of a message at its input terminal, the aggregator operator adds the message to each aggregate for which the addToAggregate condition (which will depend on the message header or body contents) evaluates to true.

Each time a message is added to an aggregate, the aggregateComplete condition is evaluated for that aggregate. If it evaluates to true, then a message is constructed from the messages it holds and is sent on the output terminal. The mapping from the messages contained in the aggregate to the message sent is specified by the aggregate operation.

If no aggregateComplete evaluates to true, then no message is sent.

#### *Constraints*

The type of content of the terminals must have a stereotype <<MessageContent>> or one of its substereotypes.

The aggregator class must have associations with one or more classes with the stereotype <<MessageAggregation>>.

A class stereotyped <<MessageAggregation>> must have addToAggregate, aggregationComplete and aggregate operations.

The addToAggregate operation of each message aggregation class must have two arguments, the first of which matches the content format of the in terminal of the aggregator operator and the second of which is a sequence of this content format, and a result of type Boolean.

The aggregationComplete operation of each message aggregation class must have a single argument whose type is a sequence of the message content format of the in terminal of the aggregator operator and a result of type Boolean.

The aggregate operation of each message aggregation class must have a single argument whose type is a sequence of the message content format of the in terminal of the aggregator operator and a result whose type matches the content format of the out terminal of the aggregator operator.

### *8.3.7 Timers*

Figure 8-8 shows the general format of the notation used to define a timer.

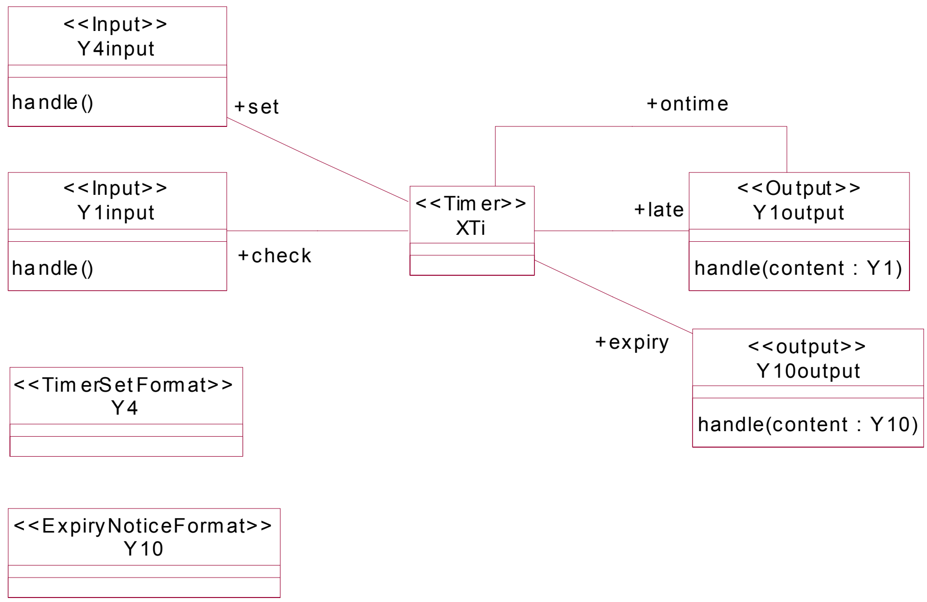



*Figure 8-8* Class diagram for prototypical timer

A timer is specified using the <<Timer>> stereotype. It processes a message on its set terminal that specifies a timer set message that contains a pair comprising a timer and a correlation condition. This gets added to the timer's list of condition pairs. When a timer receives a message from the check terminal, it looks through its list of condition pairs and sees if the message satisfies any of the correlation conditions. If so, then the timer condition is examined to see if it has been met, and, if so, the message is past onto the ontime terminal. Otherwise it is passed onto the late terminal. If it does not meet any correlation condition, it is assumed the message is on time and therefore passed onto the ontime terminal.

Whenever a timer condition from the list of condition pairs expires, an expiry notice is sent to the expiry terminal.

### *Constraints*

The input terminals must be labeled set and check. The output terminals must be labeled ontime, late, and expiry.

The content format of the check, late, and ontime terminals must be the same. This type must have stereotype <<MessageContent>> or one of its substereotypes.

The type of content of the set terminal must have a stereotype <<TimerSetFormat>>.

The type of content of the expiry terminal must have a stereotype <<ExpiryNoticeFormat>>.

## 8.3.8 Routers

Figure 8-9 shows the general format of the notation used to define a router.

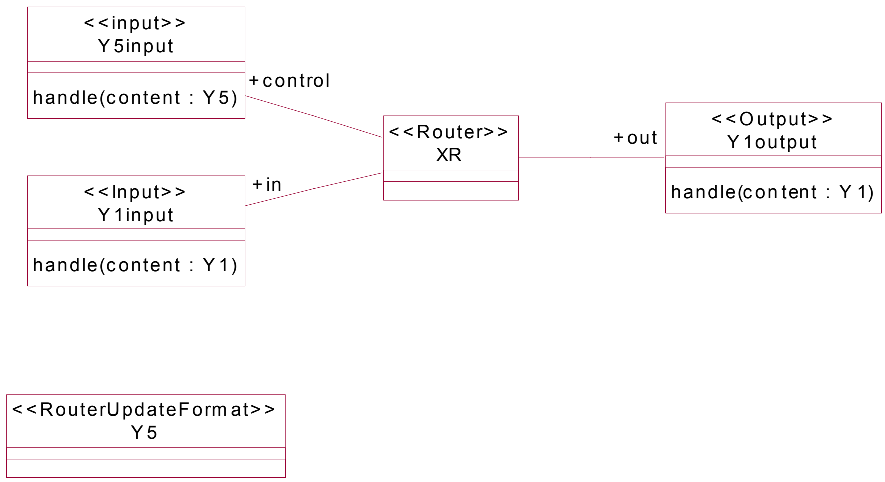



*Figure 8-9* Class diagram for prototypical router

A router is specified using the <<Router>> stereotype. When a router receives a message on its "in" terminal, it resends a copy, via its out terminal, to all terminals listed in an associated routing table. The routing table is shown as a class with stereotype <<RoutingTable>>, with a directed association from the router to it, with role name "routingTable."

A router updater can be used to make dynamic additions or removals of target terminals to or from a routing table. This can be used to model a simple publication channel for messages. A router updater is specified using the <<RouterUpdate>> stereotype, with a directed "routerUpdater" association from the router updater to a routing table. When a router updater receives a message on its "control" terminal that is in a router-update format, it performs the adds or removes given in that message on the associated routing table.

Note that, if a router has static EAILinks on its "out" terminal, then the target input terminals linked to it by those EAILinks are automatically added as the initial contents of the routing table for the router. If no dynamic updating is to be done on this initial contents (that is, no router updater will ever act on it), then it is not necessary to show the routing table explicitly in the model, and the router need not have a routingTable association.

### *Constraints*

A router must have a single input terminal labeled "in" and a single output terminal labeled "out." The type of content of the terminals of a router must be stereotyped by <<MessageContent>> or one of its substereotypes.

A router updater must have a single input terminal labeled “control” and no output terminals. The type of content of the “control” terminal of a router updater must have the stereotype <<RouterUpdateFormat>>.

A router updater must have a directed association to a class stereotyped <<RoutingTable>> with the role name routingTable.

## 8.3.9 Subscription Operators

Figure 8-10 shows the general format of the notation used to define a subscription operator.

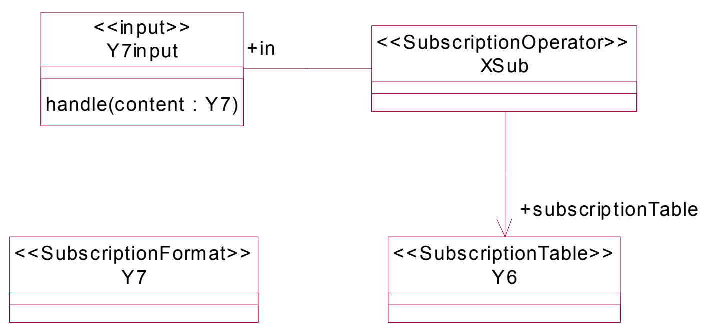



*Figure 8-10* Class diagram for prototypical subscription operator

A subscription operator is specified using the stereotype <<SubscriptionOperator>>. It expects a message of subscription format as input. This carries a subscription comprising a terminal identifier and a filter definition. When it receives one of these messages, it adds the subscription to its subscription table. A subscription message may also request subscriptions for a terminal to be canceled.

***Constraints***

The type of content of the input terminal must have a stereotype <<SubscriptionFormat>>.

There must be a directed association to a subscription table (i.e., a class with stereotype <<SubscriptionTable>>). This should be labeled subscriptionTable.

## 8.3.10 Publication Operators

Figure 8-11 shows the general format of the notation used to define a publication operator.

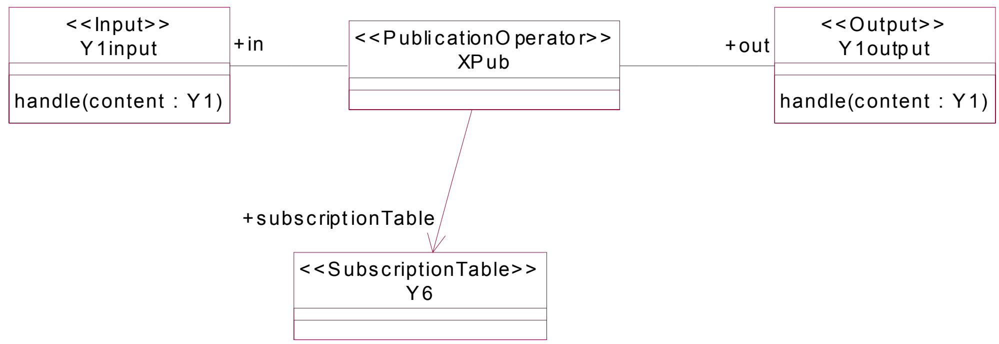



*Figure 8-11* Class diagram for prototypical publication operator

A publication operator is specified using the stereotype <<PublicationOperator>>. Messages sent to the input terminal are sent from the output terminal to each subscriber (terminal) if the message passes the filter specified by the subscription for that subscriber.

A publication operator is accompanied by at least one subscription operator when defined as part of an architecture. See Section 8.3.12.5, "Publish and Subscribe," on page 8-21 for details.

### *Constraints*

The type of content of both terminals must be the same and have a stereotype <<MessageContent>> or one of its substereotypes.

There must be a directed association to a subscription table (i.e., a class with stereotype <<SubscriptionTable>>). This should be labeled subscriptionTable.

## *8.3.11 Topic Publishers*

Figure 8-12 shows the general format of the notation used to define a topic publisher.

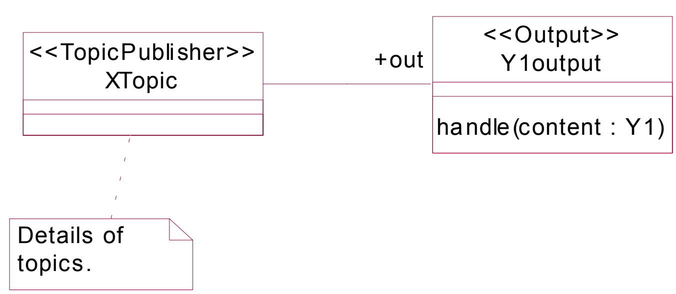



*Figure 8-12* Class diagram for prototypical topic publisher

A topic publisher is specified using the stereotype <<TopicPublisher>>. It is kind of source, which sends only sends messages to the output terminal on a set of specified topics. Details about the topics may be added as a note. The content type of the output terminal may also be an indicator of the kinds of topics published on.

Topic publishers are usually connected to the input terminal of a publication operator. See Section 8.3.12, “Compound Operators,” on page 8-14 for details.

#### *Constraints*

There is a single output terminal.

The type of content of the output terminal must have a stereotype <<MessageContent>> or one of its substereotypes.

## *8.3.12 Compound Operators*

Compound operators allow more complex message transformation and routing behavior from a (possibly nested) composition of individual operators to be modeled. Indeed any non-trivial architecture will be modeled as a compound operator whose components will be primitive or other compound operators.

Compound operators are defined using a combination of class and collaboration diagrams.

### *8.3.12.1 Class diagrams*

Figure 8-13 shows the class diagram for an example compound operator, which is specified using the stereotype <<CompoundOperator>>. The example is taken from Chapter 10.

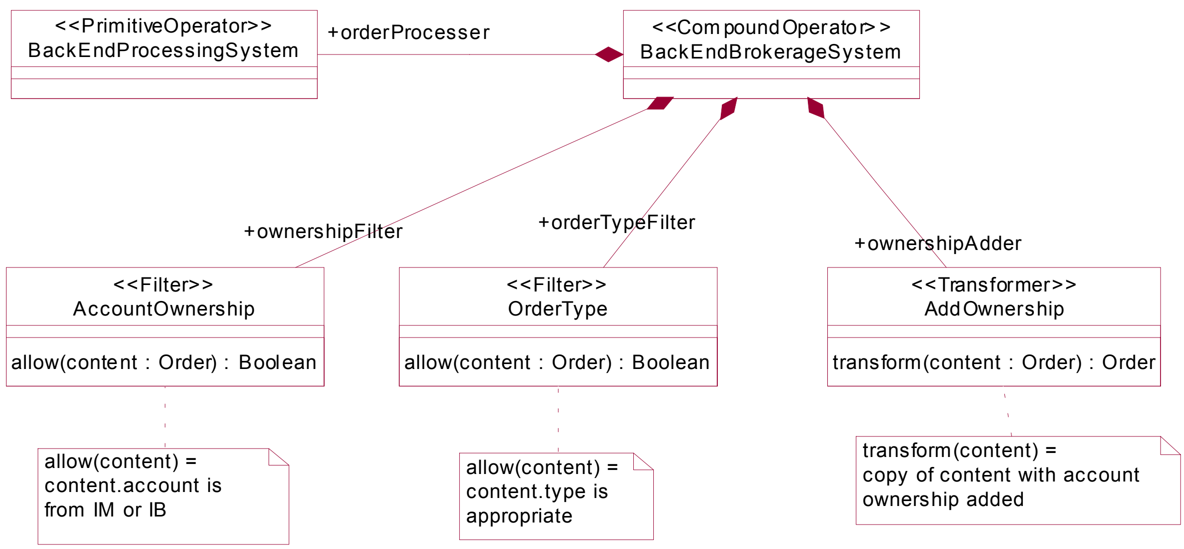



*Figure 8-13* Class diagram for example compound operator

This defines a compound operator called BackEndBrokerageSystem with three components: two filters and a transformer. The primitive operator, filters, and transformers are defined as previously discussed. Components are shown by means of a composite association targeted on a class representing an operator definition. Although the components shown here are all primitive operators, they may be compound operators, as illustrated by Figure 8-14.

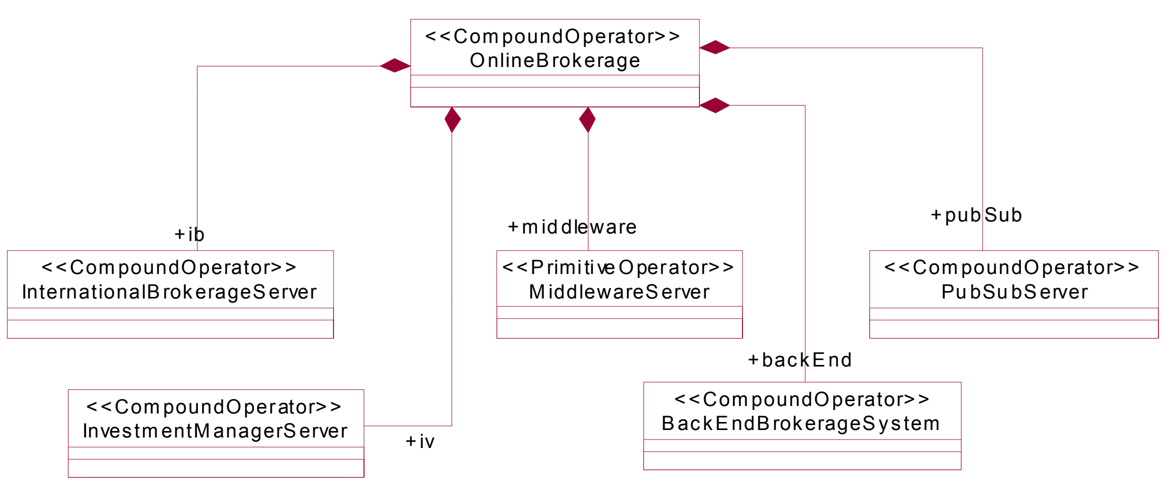



*Figure 8-14* Class diagram for a compound operator with compound components

Note, in this diagram, that one component of an OnlineBrokerage is a BackEndBrokerageSystem, which, as we have already seen, is a compound operator.

As with primitive operators, class diagrams can also be used to define the terminals of a compound operator. The terminals of BackEndBrokerageSystem are defined by Figure 8-15.

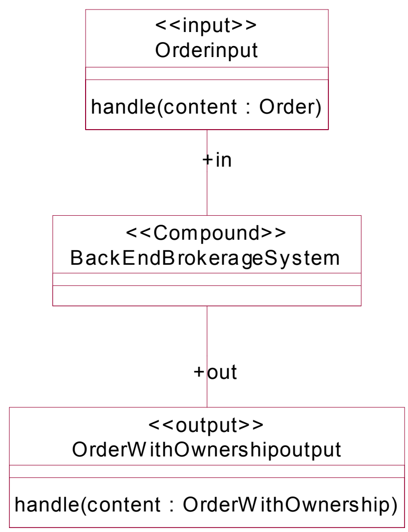



*Figure 8-15* Terminals for example of compound operator

Figure 8-15 does not show the connectivity of the components, that is, how the terminals of the components are connected together and connected to the terminals of the compound operator. A collaboration diagram is used to show the connectivity of the components.

### *8.3.12.2 Collaboration Diagrams*

The collaboration diagram corresponding to Figure 8-13 is given in Figure 8-16.

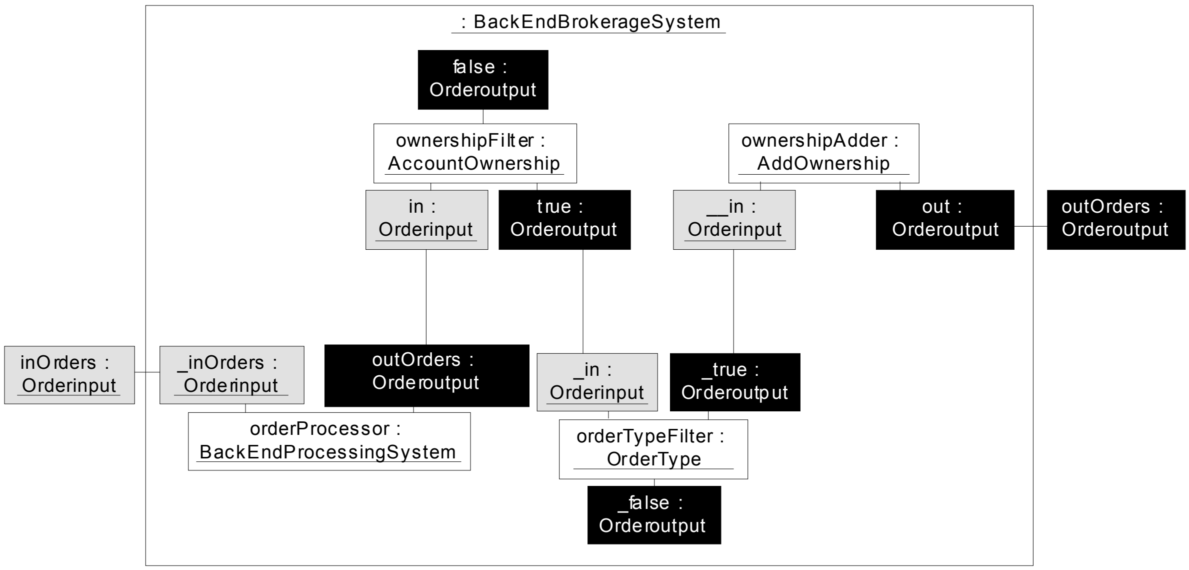



*Figure 8-16* Collaboration diagram for example compound operator

This shows:

- The components of the compound as objects contained in an object representing the compound.
- The terminals of the components (also contained in the compound), and the terminals of the compound itself (outside the compound).

The names of the objects correspond to the names of the components or terminals, as declared on the class diagram. The compound object has no name, as it represents an arbitrary operator of the compound-operator type being defined. We have used gray (or black) to distinguish input (or output) terminals from operators; this is just a convention. Connection of components is shown by connecting the terminals in an appropriate way (see Section 8.3.12.6, “Constraints,” on page 8-22 for a definition of what is appropriate)[1]. Ownership of terminals by an operator is also shown through links; the convention is to cluster terminals around their operator.

The UML 1.4 metamodel requires links to be connected to associations. In this case, the associations are redundant, but any tool strictly conforming to UML 1.4 should force the link to be associated with an association. To accommodate this, all UML for

1. Underscores on names are used to ensure uniqueness, a requirement of the tool used.

EAI models should include a class EAITerminal, from which all Terminal classes inherit. It has an association to itself with cardinality 0...* on each end, and its end names are left empty. All terminal-to-terminal links are instances of this association.

Sometimes one may wish to be explicit about whether the connection between terminals is synchronous or asynchronous. This is shown by putting a message on the link, which is marked as asynchronous or synchronous. Figure 8-17 shows the standard UML notation for this.

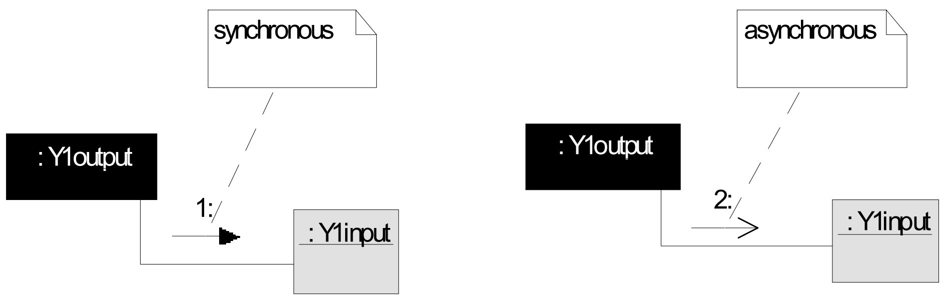



*Figure 8-17* Synchronous and asynchronous links

The arrow of the message goes in the direction of the message flow (output to input when terminals of components of a compound are connected).

### *8.3.12.3 Components of the same type*

A situation that the modeler should be aware of is the case where a compound may include two components of the same type of operator. This is illustrated by Figure 8-18 and Figure 8-19. The point to note is that there are two components of StandardIBSystem operator type (which is evident from the two associations to the StandardIBSystem class on the class diagram) and two objects of this class on the collaboration diagram.

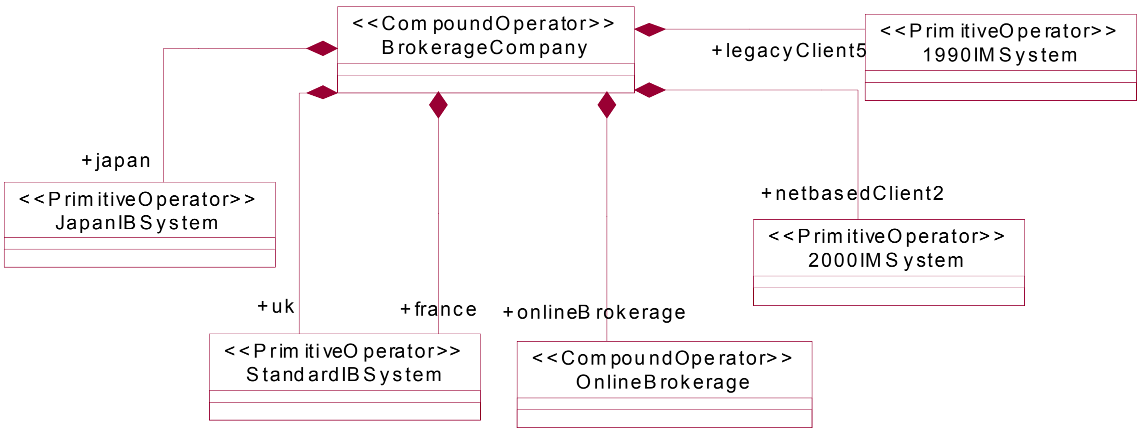



*Figure 8-18* Class diagram for example with components of same type

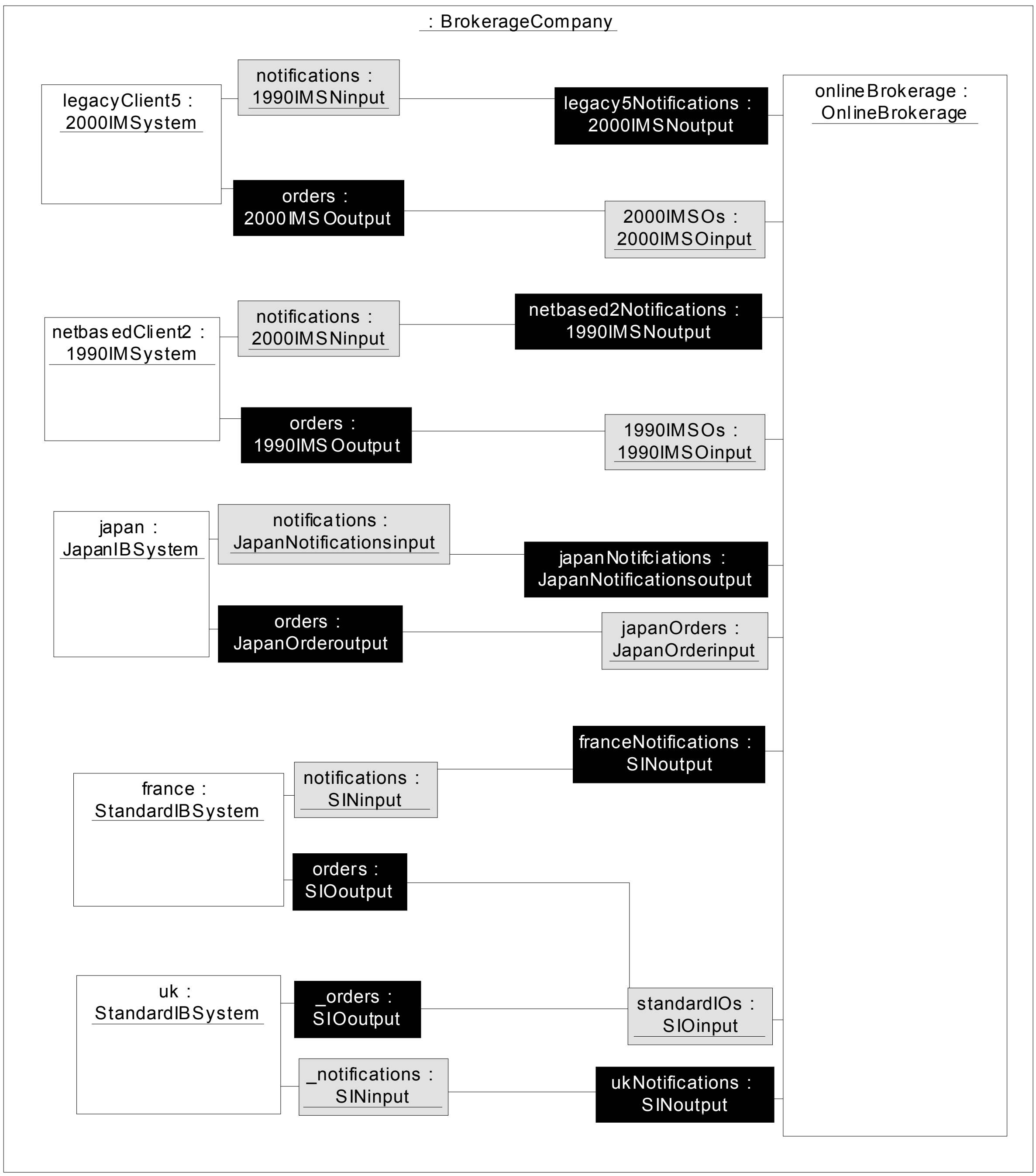



*Figure 8-19* Collaboration diagram for example with components of same type.

This example happens to illustrate the top-level definition of an EAI architecture, in this case for a brokerage company.

### 8.3.12.4 *Call and Request/Reply Adapters*

A common configuration of components is the connection of call and request/reply adapters. This is illustrated by Figure 8-20.

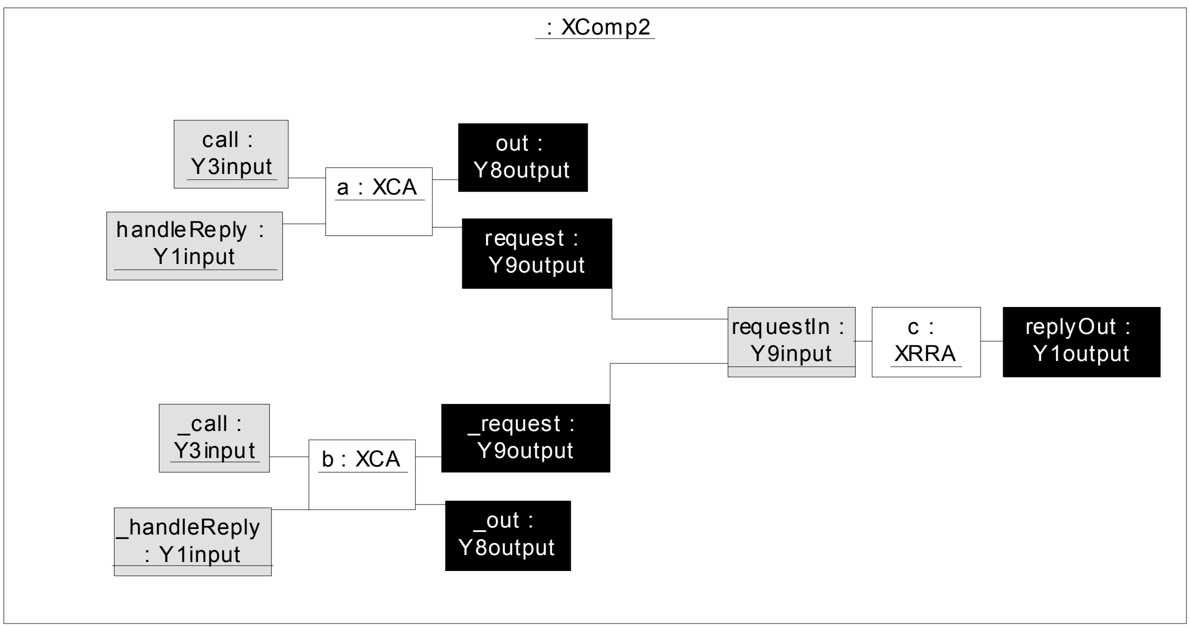



*Figure 8-20* Configuration of call and request/reply adapters

Here, two call adapters (a and b) are connected to a single request/reply adapter (c). The call adapters get information from an underlying system through their call terminals. They construct requests that are then passed on to the requestIn terminal of the request/reply adapter. This processes the request, usually by making a call to some underlying system, and then constructs a reply, which it puts on its replyOut terminal. Before sending the reply, the original request is examined to identify the terminal to which the reply must be sent (which will be the handleReply terminal for a or b, depending on which one sent the request), and this is added to the target terminals list of replyOut, just for the duration of sending the reply.

### 8.3.12.5 *Publish and Subscribe*

Another common configuration of components is the connection of publication and subscription operators. This is illustrated by Figure 8-21.

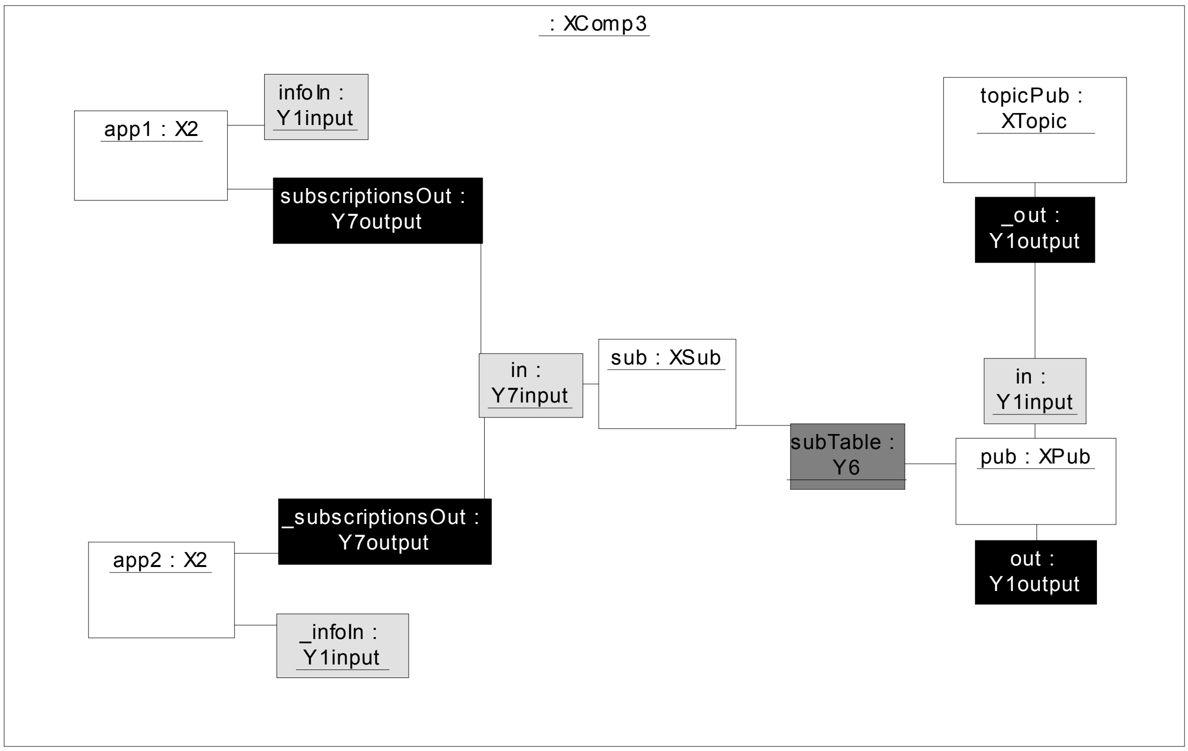



*Figure 8-21* Configuration of publication and subscription operators

A publication operator pub is fed information to publish by a topic publisher topicPub. The feed is provided by the connection of the out terminal of topicPub to the in terminal of pub. Now pub has a subscription table (subTable) that it shares with the subscription operator sub. Two applications, app1 and app2, send subscription requests to sub. The subscription requests will identify their infoIn terminals as the terminals where published information, matching the criteria of the subscriptions, should be received.

A more sophisticated (and more common) version of this example would have multiple topic publishers feeding messages to the publication operator. Then multiple publishers would share the subscription table of the subscription operator.

### *8.3.12.6 Constraints*

Only operators with stereotype <<Compound>> can have composition associations, and these must be with other operators (classes with an operator stereotype). The associations have a label but no indication of cardinality.

The type of content of the terminals must have a stereotype <<MessageContent>> or one of its substereotypes.

The class and collaboration diagrams used to notate a compound operator must be consistent. This means:

- Names of terminal objects must match the labels on terminal associations on the class diagram. The types of the object must correspond to the terminal classes defined in the class diagram.
- Names of component operator objects must match the labels on the composition associations on the class diagram. The types of the objects must correspond to the operator classes at the target of those associations as defined on the class diagram.

On the collaboration diagram, only output terminals may be connected to input terminals of other components. Input (output) terminals of the compound operator may only be connected to input (output) terminals of components.

The content type handled by terminals must be the same for any two terminals connected together on the collaboration diagram.

## 8.4 Adapters

### 8.4.1 Source Adapters

Figure 8-22 shows the general format of the notation used to define a source adapter, which is represented by a class with stereotype <<SourceAdapter>>. A source adapter is an operator that obtains information from a system (e.g., vendor-supplied package or legacy application system), where that information might not be in a message content format, translates it into message content of a given output type and then sends out a message with that content.

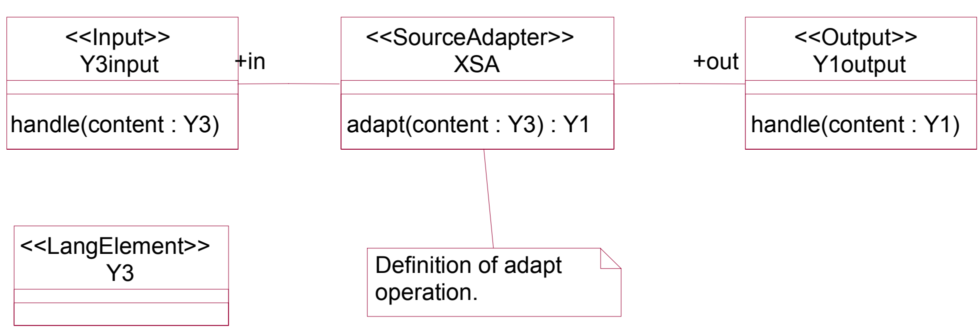



*Figure 8-22* Class diagram for prototypical source adapter

When using a source adapter as a component of a compound operator (see Section 8.3.12, “Compound Operators,” on page 8-14), it is usually the case that its input terminal will not be connected to any other terminals. How information gets placed on that terminal is left unstated, since the internals of an application are out of scope for EAI modeling.

### Constraints

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the adapt operation.

The type of content of the output terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the input terminal must have a stereotype <<LangElement>> or one of its substereotypes.

## 8.4.2 Target Adapters

Figure 8-23 shows the general format of the notation used to define a target adapter, which is represented by a class with stereotype <<TargetAdapter>>. A target adapter is an operator that accepts messages and translates them into information for a system (e.g., vendor-supplied package or legacy application system), where that information might not be in a message content format.

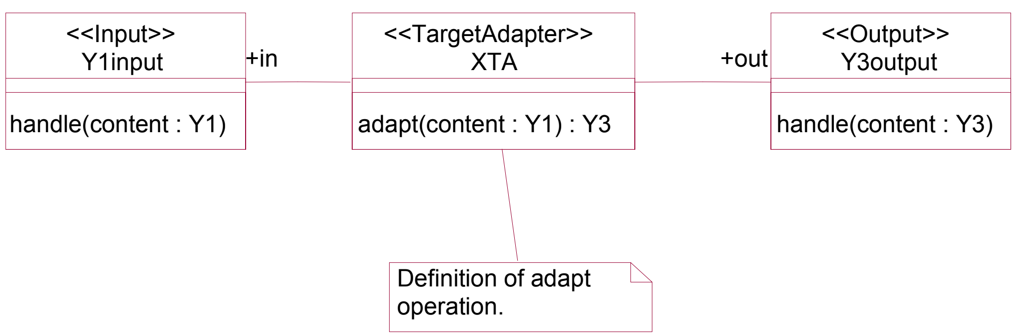



*Figure 8-23* Class diagram for prototypical target adapter

When using a target adapter as a component of a compound operator (see Section 8.3.12, “Compound Operators,” on page 8-14), it is usually the case that its output terminal will not be connected to any other terminals. What happens to information after it leaves that terminal is left unstated, since the internals of an application are out of scope for EAI modeling.

### Constraints

The content format of the input and output terminals must match the format of the parameter and result, respectively, of the adapt operation.

The type of content of the input terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the output terminal must have a stereotype <<LangElement>> or one of its substereotypes.

## 8.4.3 Call Adapters

Figure 8-24 shows the general format of the notation used to define a call adapter.

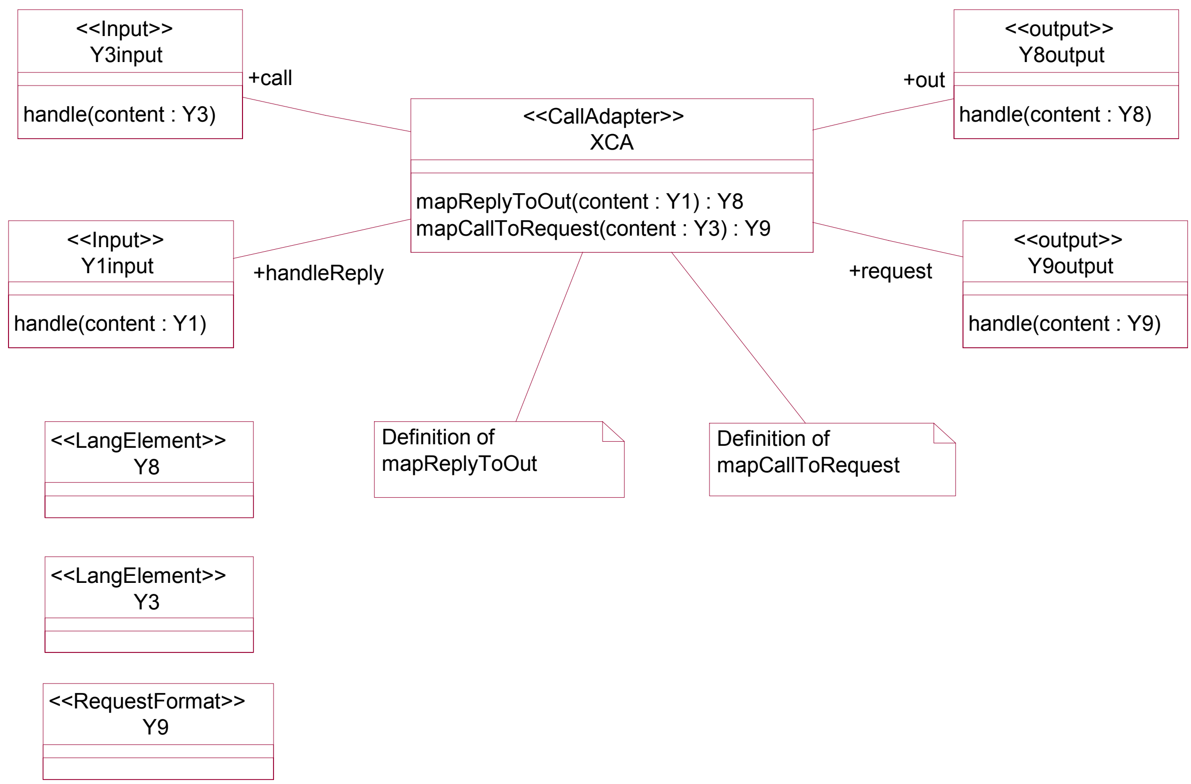



*Figure 8-24* Class diagram for prototypical call adapter

A call adapter is invoked synchronously by an application that wishes to make use of a service (made available via a server) that can respond to a request message and send a response message back to the service requester. It accepts a call (which is not in a standard message format) on its call terminal and maps that call to a request message, which it sends to the request terminal. On receipt of a reply from the handleReply terminal, it maps that reply to a format understood by the application and places the result of the mapping on the out terminal.

A call adapter is used in conjunction with a request/reply adapter. See Section 8.3.12.4, “Call and Request/Reply Adapters,” on page 8-21 for details.

### *Constraints*

The input terminals must be labeled call and handleReply, and the output terminals out and request.

The type of content of call and request must match the type of the parameter and result, respectively, of the mapCallToRequest operation.

The type of content of handleReply and out must match the type of the parameter and result, respectively, of the mapReplyToOut operation.

The type of content of the handleReply terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of the content of the call and out terminals must have a stereotype <<LangElement>> or one of its substereotypes. The type of content of the request terminal must have a stereotype <<RequestFormat>>.

### 8.4.4 Request/Reply Adapters

Figure 8-25 shows the general format of the notation used to define a request/reply adapter.

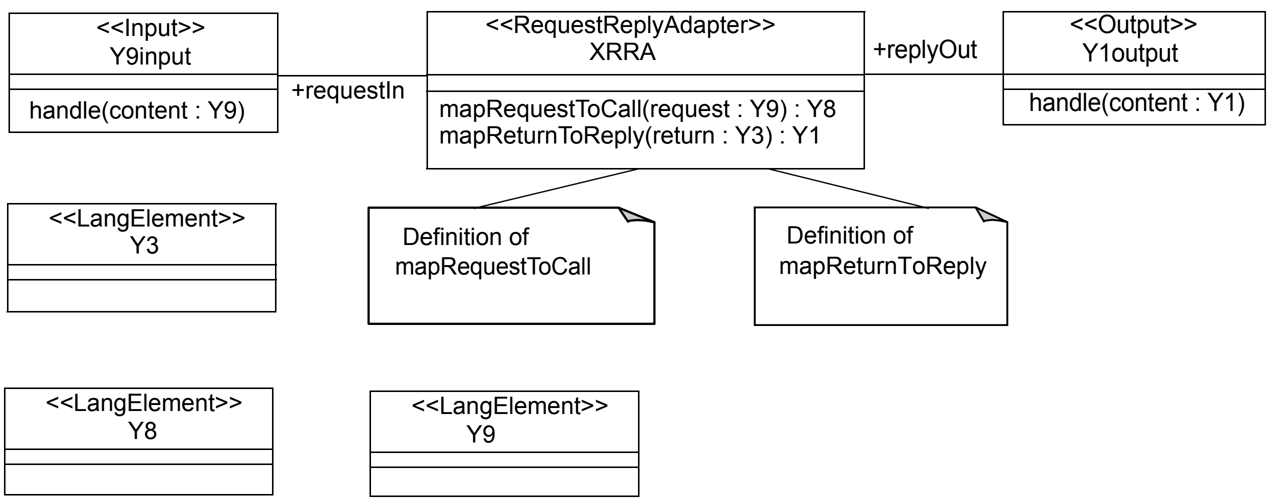



*Figure 8-25* Class diagram from prototypical request/reply adapter

A request/reply adapter receives a request (from a call adapter) that contains both a terminal identifier and some other content. The mapRequestToCall operation extracts the information content of the request and converts it to a format suitable for passing to some underlying system. The mapReturnToReply operation takes the information returned from the system and constructs a message that is placed on the output terminal, but only after the terminal identifier in the original request has been added to the target list of its replyOut terminal. When the message has been sent, the terminal identified in the request message is removed from the target set of replyOut.

Note that any terminal permanently connected to the replyOut terminal will have replies of all requests broadcast to it.

A request/reply adapter is used in conjunction with a call adapter. See Section 8.3.12.4, "Call and Request/Reply Adapters," on page 8-21 for details.

### Constraints

The input terminal must be labeled requestIn, and the output terminal replyOut.

The type of content of requestIn and replyOut must match the type of the parameter of mapRequestToCall and the result of mapReturnToReply, respectively.

The type of content of the replyOut terminal must have a stereotype <<MessageContent>> or one of its substereotypes. The type of content of the requestIn terminal must have a stereotype <<RequestFormat>>. The type of the result of mapRequestToCall and the parameter of mapReturnToReply must have a stereotype of <<LangElement>> or one of its substereotypes.

# 8.5 Sources and Sinks

## 8.5.1 Sources and Queued Sources

Figure 8-26 shows the general format of the notation used to define a source, which is represented by a class with stereotype <<Source>>. A source is an operator that delivers message content to an output terminal. How that message content is constructed, or where it comes from, is not stated.

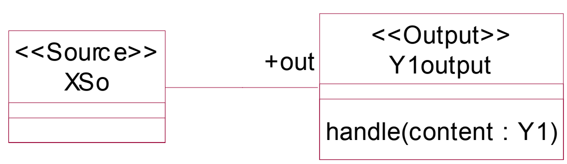



*Figure 8-26* Class diagram for prototypical source

A queued source is a source that has a <<Queue>> resource. It is identified by the stereotype <<QSource>>, as illustrated by Figure 8-27.

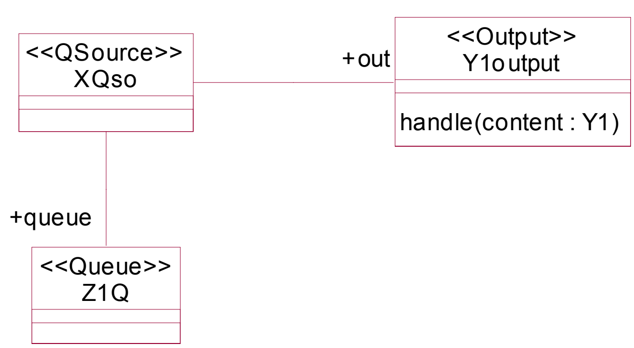



*Figure 8-27* Class diagram for prototypical queued source

***Constraints***

The type of content of the output terminal must have a stereotype <<*MessageContent*>> or one of its substereotypes.

For queued sources, there must be a directed association to a queue resource (i.e., a class with stereotype <<*Queue*>>). This should be labeled *queue*.

## *8.5.2 Sinks and Queued Sinks*

Figure 8-28 shows the general format of the notation used to define a sink, which is represented by a class with stereotype <<*Sink*>>. A *sink* is an operator that receives message content from an input terminal. What happens to that content thereafter is left unsaid.

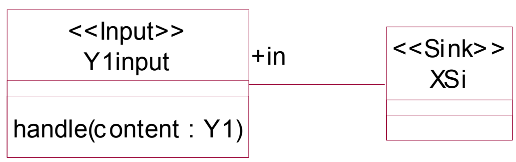
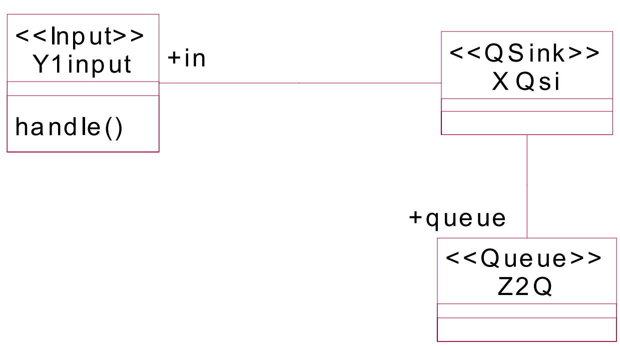



*Figure 8-28* Class diagram for prototypical sink

A queued sink is analogous to a queued source, and is identified by the stereotype <<QSink>>.




***Constraints***

The content format of the input terminal must have a stereotype <<MessageContent>> or one of its substereotypes.

For queued sinks, there must be a directed association to a queue resource (i.e., a class with stereotype <<Queue>>). This should be labeled queue.

## 8.6 Resources

Resources are things that operators use to do their job, but which are not themselves operators. The specific resources declared in this profile are databases, queues, and subscription tables.

Resources are defined as classes with stereotype <<Resource>> or one of its substereotypes: <<Database>>, <<Queue>>, and <<SubscriptionTable>>.

The use of a resource by an operator is indicated, in the class diagram defining that operator, by a directed association from the operator to the resource. See Section 8.3.2, "Transformers and Database Transformers," on page 8-4 and "A router updater must have a directed association to a class sterotyped <<RoutingTable>>, with the role name routingTable," on page 8-12 for examples.

When operators with resources are used as part of a compound, they may share a resource. This is shown by adding an object of the resource class and connecting the sharing operators to it with a link. See Section 8.3.12.5, "Publish and Subscribe," on page 8-21 for an example.

## 8.7 Message Formats

### 8.7.1 MessageContent Core

The data contained in a message is its MessageContent. Messages are defined using ordinary UML class modeling mechanisms. However, message content classes are restricted to represent transmittable data structures.

The model for messages is that they may contain one or more parts, each of which may have its own header part. The header contains information used by the messaging infrastructure to control how it deals with the message. Each message part may also have a body section, which contains the application data. Message parts may be nested.

Both the header and the body may contain nested structures of primitive message elements.

We formalize these restrictions using the UML stereotypes given in Table 8-1. A class of the <<MessageContent>> stereotype represents a serialized message. To reflect the ordering of the parts of a message, there are additional constraints:

1. All associations are ordered with respect to each other.

2. Associations of multiplicity greater than one are ordered.

For example, a message header usually occurs in a message part before the message content.

*Table 8-1* Stereotype specification for message content description

| Stereotype | Parent | Tags | Constraints | Description |
|---|---|---|---|---|
| MessageContent | N/A | domainformat | May only have containment associations with classes of stereotype <<MessagePart>> or <<ComposedMessagePart>> | Top level for describing messages (such as a MIME envelope). |
| MessagePart | N/A | NA | May only be composed by a class of stereotype <<MessageContent>>May contain a 'header' association with a class of stereotype <<MessageElement>>May contain a 'body' association with a class of stereotype <<MessageElement>> | Used to describe 'large scale' message structuring (such as MIME parts). |
| ComposedMessagePart | MessagePart | NA | May have associations with classes of stereotype <<MessagePart>> and of stereotype <<ComposedMessagePart>> | Used to describe nested message parts. |
| LangElement | NA | NA | | Models message headers, message bodies and their content. |

Figure 8-29 shows an example of a content class with two data items, an integer and a string. These simple message parts have been rendered as attributes of the owning SimpleContent class. This is recommended in order to allow compact representation of simple message types.

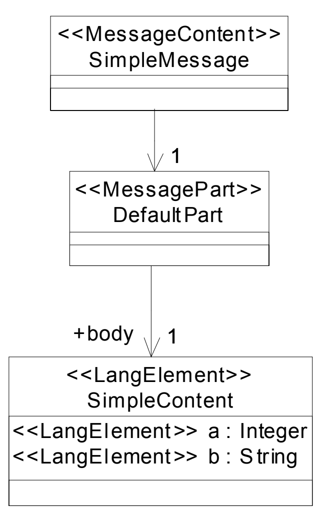



*Figure 8-29* A simple message content class

More complicated message-content structures can be created using composition, as is shown in Figure 8-30. This models a message that has a single part. The message has as its header a string, while the message body is a table of addresses. This table has a single integer, records, that is a count of the records in the message.

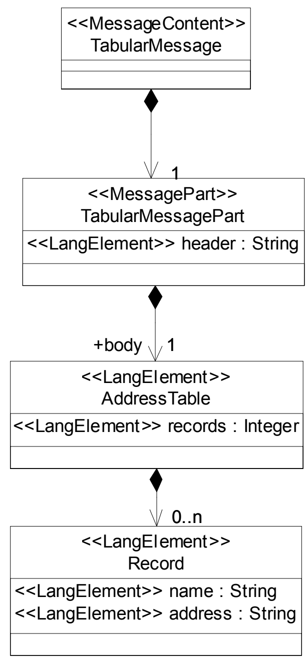



*Figure 8-30* A model of a message containing a table

## *8.7.2 Basic MOM Message Structure*

The stereotypes given in the preceding section provide the framework to allow messages to be specified, but they do not cover commonly occurring concepts supported by message oriented middleware (MOM) products.

In this section we add the basic concept of an exception message, a message sent by the messaging infrastructure when a fault occurs in the processing of a message. We also define a MOMHeader, which can specify an exception target (the location to which a message should be sent in the event of an exception) and a reply target, and it can identify the kind of message being sent.

*Table 8-2* Stereotype specification for MOM structure

| **Stereotype** | **Parent** | **Tags** | **Constraints** | **Description** |
|---|---|---|---|---|
| MOMHeader | MessageElement | NA | May have an association 'replyTo' with a <<MessageElement>> class that specifies a reply target and another 'exceptionTarget' with a <<MessageElement>> class that specifies an exception target. | Stereotype to capture common MOM header information. |
| ExceptionNotice | MessageContent | NA | May have a message part containing the header and body of the message that caused the fault. | Message sent by the MOM infrastructure if a fault occurs while processing a message. |

### *8.7.2.1 ExceptionNotice*

Figure 8-31 illustrates the usage of the ExceptionNotice stereotype. In this example, we have defined a class MOMException, which models the message content of an exception message created by a MOM system after a fault has occurred. MOMException contains two associations to classes that conform to the MessagePart stereotype:

- originalMessage: an association to a class that models the content of the message that caused the exception. In this case, the original message had just one message part. If the original message had contained several parts, it would be possible to model originalMessage as a class that conforms to the ComposedMessagePart stereotype.
- exceptionInformation: an association to a message part that contains only exception header information. The exception header holds information that identifies the exception type and a string that describes the exception.

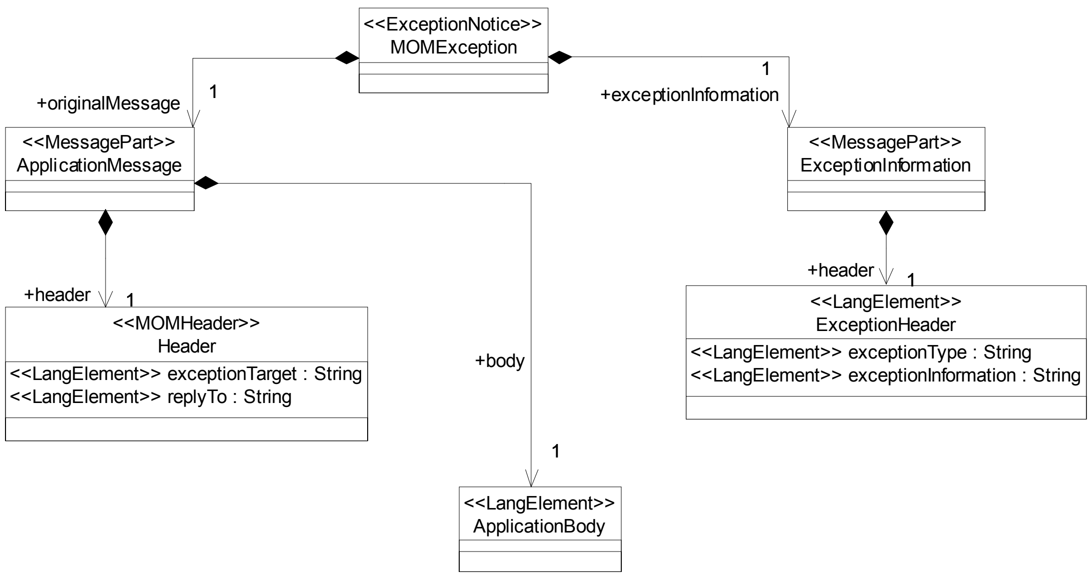



*Figure 8-31* Example of the use of the ExceptionNotice and MOMHeader stereotypes

### *8.7.2.2 MOMHeader*

The MOMHeader stereotype demands that a message header must identify the following elements, but does not dictate how they are represented in the message:

- replyTo: a means of identifying a location to send a reply message to.
- exceptionTarget: a means of identifying a location to send an exception notice in the event of a fault occurring in the processing of a message.

Figure 8-32 demonstrates an example of the use of the MOMHeader stereotype. In this case, the domain and format are both identified using strings, and the exceptionTarget and replyTo header content are specified using the MOMEndpointSpec class. In a particular MOM implementation, this information should allow an EAI terminal to be identified.

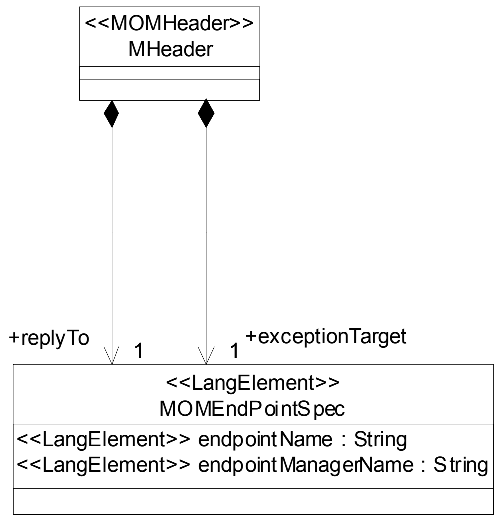



*Figure 8-32* Example of the use of the MOMHeader stereotype

## 8.8 Mapping with Metamodel

The mapping with the metamodel is summarized by a series of tables, which are organized below into sections corresponding to the four main parts of the profile: terminals, operators, resources, and message formats.

These tables are based on the approach specified in UML 1.4. for defining stereotypes for use in a profile. We have extended them to show the mapping to the EAI metamodel. Thus the tables also serve to summarize the stereotypes used in the profile.

In addition to the tables, we have detailed important mapping constraints that dictate how information associated with an instance of an EAI metaclass is related to information associated with an instance of the stereotyped UML base class. These are listed below the relevant tables.

The mapping constraints should be distinguished from constraints that apply to the use of the profile itself (e.g., the use of a particular stereotype). Those are defined in the section describing that aspect of the profile.

### 8.8.1 Terminals

*Table 8-3* Mapping of terminals

| EAI Metaclass | Base class | Stereotype | Parent | Description & constraints |
|---|---|---|---|---|
| EAITerminal | Core::Association | | | See Section 8.2 |
| | Core::Class | Input or Output | | See Section 8.2 |
| EAIQueuedInputTerminal | Core:: Association | | | See Section 8.2 |
| | Core::Class | QInput | Input | See Section 8.2 |
| EAIQueuedOutputTerminal | Core:: Association | | | See Section 8.2 |
| | Core::Class | QOutput | Output | See Section 8.2 |

#### *Mapping Constraints*

***EAITerminal***

1. This mapping is valid only for terminals that belong to operators that define types.
2. The association is sourced on the class corresponding to the operator to which the terminal belongs; it is targeted on the class identified with the terminal.
3. The handle operation of the class must have a parameter of a type corresponding to the type of the parameter associated with the terminal.
4. Different terminals may map to the same class (but not the same association).
5. The name of the terminal is the name of the target end of the association.
6. The stereotype of the class corresponds to the value of the terminalKind attribute of the terminal.

***EAIQueuedInputTerminal and EAIQueuedOutputTerminal***

There are no additional constraints.

### 8.8.2 Operators

Operators and terminals in the metamodel are used in two roles. Firstly they are used to define types and parameters; secondly they are used to define the connectivity of a compound operator in its role in defining a type. The mapping of operators has been split into two parts, reflecting the two different roles. The first part deals with all operators, except compound operators. The second part deals with compound operators, which, as suggested above, requires a second mapping of operators and terminals to be defined.

*Table 8-4* Mapping of operators (except compound)

| EAI Metaclass | Base class | Stereotype | Parent | Description & constraints |
|---|---|---|---|---|
| EAIPrimitiveOperator | Core::Class | PrimitiveOperator | | See Section 8.3.1 |
| EAITransformer | Core::Class | Transformer | PrimitiveOperator | See Section 8.3.2 |
| EAIDBTransformer | Core::Class | DBTransformer | Transformer | See Section 8.3.2 |
| EAIFilter | Core::Class | Filter | PrimitiveOperator | See Section 8.3.3 |
| EAIStream | Core::Class | Stream | PrimitiveOperator | See Section 8.3.4 |
| EAIPostDater | Core::Class | PostDater | Stream | See Section 8.3.5 |
| EAIAggregator | Core::Class | Aggregator | PrimitiveOperator | See Section 8.3.6 |
| EAISubscriptionOperator | Core::Class | SubscriptionOperator | PrimitiveOperator | See Section 8.3.9 |
| EAIPublicationOperator | Core::Class | PublicationOperator | PrimitiveOperator | See Section 8.3.10 |
| EAITopicPublisher | Core::Class | TopicPublisher | PrimitiveOperator | See Section 8.3.11 |

### *Mapping Constraints*

#### *EAIPrimitiveOperator*

7. This mapping is only valid for primitive operators defining a type (not ones used to show connectivity of components).
8. The name of operator (and hence the type that the operator defines) is the name of the class.
9. There must be an association on the class diagram corresponding to each terminal of the primitive operator.

#### *EAITransformer*

10. The transformation mapping of the operator maps to the operation transform, in the class corresponding to the operator.

#### *EAIDBTransformer*

11. The database resource maps to the database association sourced on the class corresponding to the operator.

#### *EAIFilter*

12. The filterCondition of the operator maps to the allow operation in the corresponding class.

#### *EAIStream*

13. The emissionCondition of the operator maps to the emit operation in the corresponding class.

***EAIPostDater***

14. The timerMapping of the operator corresponds to the setTimingCondition operation in the corresponding class.

***EAIAggregator***

15. The aggregateComplete condition of each EAIMessageAggregation of the operator corresponds to the aggregateComplete operation in the corresponding <<MessageAggregation>> class.

16. The addToAggregate condition of each EAIMessageAggregation of the operator corresponds to the addToAggregate operation in the corresponding <<MessageAggregation>> class.

17. The aggregationMapping of each EAIMessageAggregation of the operator corresponds to the aggregate operation in the corresponding <<MessageAggregation>> class.

***EAISubscriptionOperator and EAIPublicationOperator***

18. The subscriptionTable resource maps to the subscriptionTable association sourced on the class corresponding to the operator.

***TopicPublisher***

There are no further constraints.

A compound operator utilizes a graph of operators, terminals, and resources to define the connectivity of its components. This is exposed by the mapping defined in the table below.

*Table 8-5* Mapping of compound operator

| EAI Metaclass | Base class | Stereotype | Parent | Description & constraints |
|---|---|---|---|---|
| EAICompoundOperator | Core::Class | CompoundOperator | | See Section 8.3.12 |
| | CommonBehavior::Object | | | See Section 8.3.12 |
| EAITimer | Core::Class | Timer | CompoundOperator | See Section 8.3.7 |
| EAIRouter | Core::Class | Router | CompoundOperator | See Section 8.3.8 |
| EAIPrimitiveOperator (and subclasses) | Core::Association, CommonBehavior::Object | | | See Section 8.3.12 |
| EAICompoundOperator (and subclasses) | Core::Association, CommonBehavior::Object | | | See Section 8.3.12 |
| EAIResource (and subclasses) | CommonBehavior::Object | | | See Section 8.3.12 |
| EAITerminal (and subclasses) | CommonBehavior::Object | | | See Section 8.3.12 |
| EAILink | CommonBehavior::Link | | | See Section 8.3.12 |

### Mapping Constraints

#### EAICompoundOperator

19. This mapping is only valid for compound operators defining a type (not ones used to show connectivity of components).

20. The name of operator (and hence the type that the operator defines) is the name of the class.

21. There must be an association on the class diagram corresponding to each terminal of the compound operator.

22. On the class-diagram part of the definition of the compound operator, there must be an association for each component operator.

23. The object is unnamed on the collaboration diagram defining the connectivity of the compound's components, and it contains all objects corresponding to the component operators and their terminals.

24. On the collaboration diagram, the objects corresponding to the terminals of the operator appear outside the object corresponding to the operator.

#### EAITimer and EAIRouter

25. Exceptionally, the components of these operators are not exposed in the profile. Therefore they do not have collaboration diagrams associated with them, and they do not map to objects.

#### EAIPrimitiveOperator (and subclasses), EAICompoundOperator (and subclasses)

26. This mapping is only valid for operators that are used in the role of defining the components of a compound operator. That is, they do not define a type, and they are owned by a compound operator (one of its nodes).

27. The association must be a composite association. The name of the part end corresponds to the name of the operator. The association is sourced on the class corresponding to the compound operator of which the operator in question is a part, and targeted on the class corresponding to the operator that defines the type of the operator in question.

28. The name of the object corresponds to the name of the operator. The type of the object is the class that corresponds to the operator that defines the type of the operator in question.

29. There must be an object corresponding to each terminal of the operator, and this must be linked to the object corresponding to the operator.

30. The object corresponding to the operator in question may be linked to an object corresponding to a resource, if the operator that defines the type of the operator in question is associated with a resource. The type of the resource object is the class corresponding to the resource.

*EAIResource (and subclasses)*

31. This mapping is only valid if the resource is associated with an operator used in the role of defining a component of a compound.

*EAITerminal*

32. The name of the object is the name of the terminal. The type of the object is the class corresponding to the terminal that defines the parameter associated with the terminal.

*EAILink*

33. The (UML) link must connect the objects associated with terminals that the (EAI) link connects.

34. The (UML) link has no message if the value of the synchronization attribute of the (EAI) link is unspecified. It has a synchronous (asynchronous) message if the value of that attribute is synchronous (asynchronous). The direction of the message is from the object corresponding to the source of the (EAI) link, to the object corresponding to the target of the (EAI) link.

## 8.8.3 Adapters

*Table 8-6* Mapping of adapters

| EAI Metaclass | Base class | Stereotype | Description & constraints |
|---|---|---|---|
| EAISourceAdapter | Core::Class | SourceAdapter | See Section 8.4.1 |
| EAITargetAdapter | Core::Class | TargetAdapter | See Section 8.4.2 |
| EAICallAdapter | Core::Class | CallAdapter | See Section 8.4.3 |
| EAIRequestReplyAdapter | Core::Class | RequestReplyAdapter | See Section 8.4.4 |

### Mapping Constraints

*EAISourceAdapter and EAITargetAdapter*

35. The internalToMessage (resp. messageToInternal) mapping for the operator corresponds to the adapt operation in the corresponding class.

*EAICallAdapter*

36. The callToRequestMapping of the operator corresponds to the mapCallToRequest operation in the corresponding class.

37. The replyToOutMapping of the operator corresponds to the mapReplyToOut operation in the corresponding class.

***EAIRequestReplyAdapter***

38. The requestToCallMapping of the operator corresponds to the mapRequestToCall operation in the corresponding class.

39. The returnToReplyMapping of the operator corresponds to the mapReturnToReply operation in the corresponding class.

## *8.8.4 Sources and Sinks*

*Table 8-7* Mapping of Sources and Sinks

| EAI Metaclass | Base class | Stereotype | Description & constraints |
|---|---|---|---|
| EAISource | Core::Class | Source | See Section 8.3.10 |
| EAIQueuedSource | Core::Class | QSource | See Section 8.3.10 |
| EAISink | Core::Class | Sink | See Section 8.3.11 |
| EAIQueuedSink | Core::Class | QSink | See Section 8.3.11 |

***Mapping Constraints***

***EAISource, EAIQueuedSource, EAISink, EAIQueuedSink***

There are no further constraints.

## *8.8.5 Resources*

*Table 8-8* Mapping of resources

| EAI Metaclass | Base class | Stereotype | Parent | Description & constraints |
|---|---|---|---|---|
| EAIResource | Core::Association | | | See Section 8.6 |
| | Core::Class | Resource | | See Section 8.6 |
| EAIDatabase | Core:: Association | | Resource | See Section 8.6 |
| | Core::Class | Database | Resource | See Section 8.6 |
| EAIQueue | Core:: Association | | Resource | See Section 8.6 |
| | Core::Class | Queue | Resource | See Section 8.6 |
| EAISubscriptionTable | Core:: Association | | Resource | See Section 8.6 |
| | Core:: Class | SubscriptionTable | Resource | See Section 8.6 |

### *8.8.5.1 Mapping Constraints*

40. The name of the resource maps to the name of the target end of the association.

41. The source of the association is the class corresponding to the operator associated with the resource.

42. The target of the association must be a class with a stereotype corresponding to the name of the (metamodel concrete) class of the resource.

### 8.8.6 Message Formats

*Table 8-9* Mapping of formats

| EAI Metaclass | Base Class | Stereotype | Parent | Description & constraints |
|---|---|---|---|---|
| EAIMessageContent | Core::Class | MessageContent | | See Section 8.7.1 |
| EAIMessagePart | Core::Class | MessagePart | | See Section 8.7.1 |
| EAIComposedMessagePart | Core::Class | ComposedMessagePart | MessagePart | See Section 8.7.1 |
| TDLangElement | Core::Class | LangElement | | See Section 8.7.1 |
| EAIHeader | Core::Class | MOMHeader | MessageElement | See Section 8.7.2 |
| EAIExceptionNotice | Core::Class | ExceptionNotice | MessageContent | See Section 8.7.2 |

#### *Mapping Constraints*

##### *TDLangElement*

43. Composed types will map to a TDLangElement with a TDLangComposedType. See Section 7.3.4.4, “TDLangElement,” on page 7-8.

# Activity Modeling 9

Messages are produced as a result of business events occurring in enterprise applications. The sequence of these events and the resulting message flows across system boundaries is defined in the overall system integration process. This section describes a profile for modeling EAI processes using activity graphs. These models can subsequently be refined to realize the functionality specified using the stereotypes defined in Chapter 8.

## 9.1 Modeling Integration Processes

Chapter 8 describes a profile for defining the collaborations necessary for application integration. It may be characterized as a profile for designing integrations. Many application-integration developers also adopt a process-oriented approach where the initial artifact is a definition of the business process, end-to-end, which is to be integrated. Of course such a process definition will encompass many integration points, each of which will need to be implemented. The value of the process view is to establish the requirement in a form that is understandable and verifiable by the business users. In this sense it is a requirements or analysis view and exists at a higher level of abstraction than the collaboration-based definitions of the previous chapter.

Whilst for any particular implementation approach it should be possible to map the analysis model onto the design model, it is beyond the scope of this specification to do so. In a sense, it would be pre-empting the development process. We consider a general formal mapping - with enforcement of a level of detail capable of formal mapping - to be inappropriate, since different practitioners have different approaches.

## 9.2 An Integration Process Scenario

Integration processes contain control flow and message flow aspects. Message flow is fundamental in EAI processes, as message-based integration is at the core of the problem domain.

This section introduces the profile elements required to define such models by means of an example scenario. Variants of the scenario are discussed. Some illustrate the capability of the profile to support high levels of abstraction, such as might be preferred for communicating with business users; others illustrate how the profile can be used to define more detail.

### 9.2.1 The Exchange Process

The scenario we have chosen is a collaborative business-to-business example, where Buyers and Sellers negotiate a transaction via an online Exchange. The overall process is represented in the activity graph in Figure 9-1. Annotations have been added (as parameters on transitions) to represent additional information about required operations (such as transformations) and to identify implementation details (such as queues). This is an example of where different practitioners might choose to capture this information at this level or may choose to omit it. The intention is that the profile is capable of representing it if required.

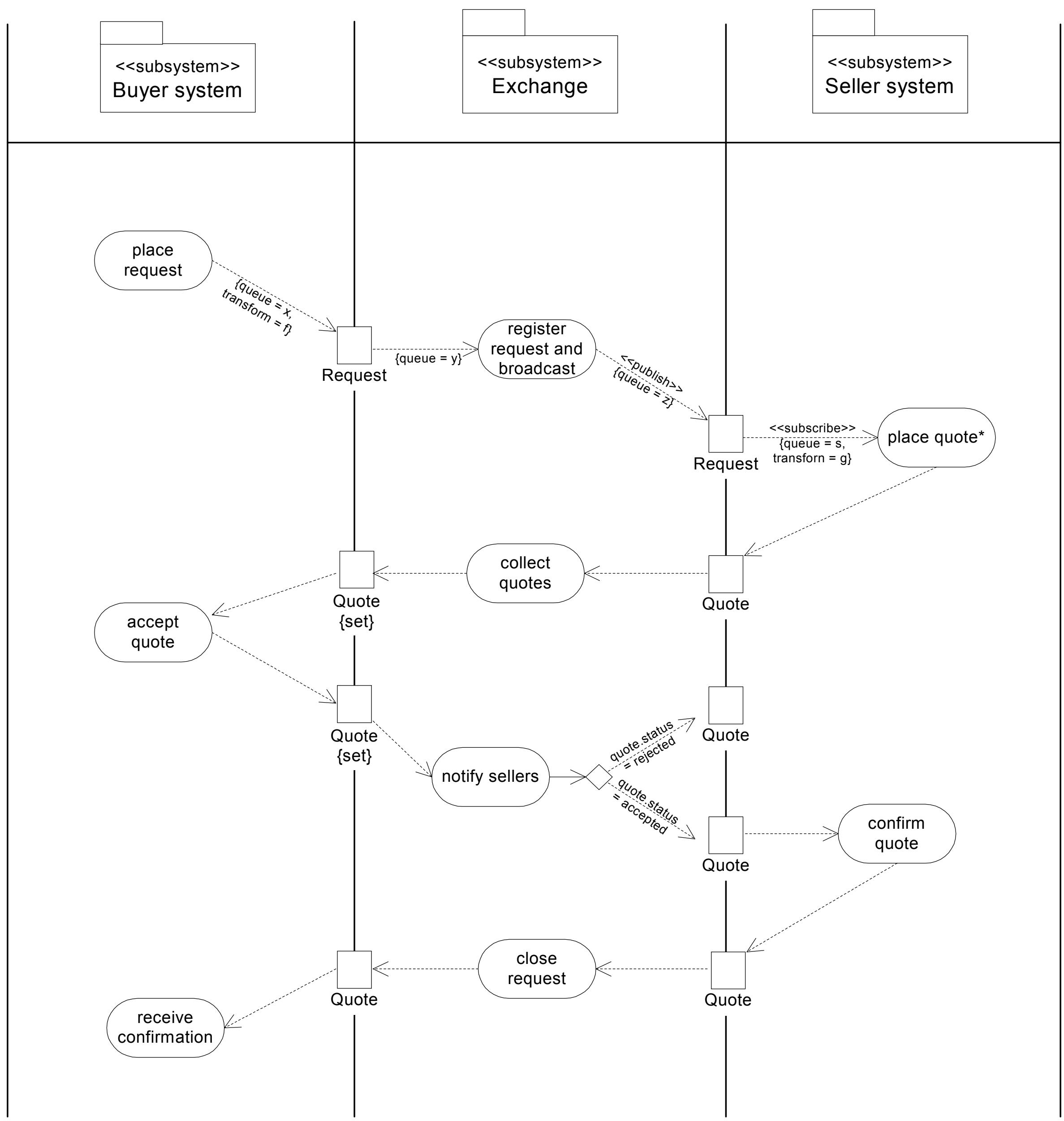



*Figure 9-1* Basic way of modeling message based integration with Activities (Exchange example)

In this example, activities represent the legacy applications that consume and/or raise business events. The existence of connectors to detect and publish the events and to interpret these events for the legacy applications is implicit at this level of abstraction.

## 9.2.2 Modeling message flow explicitly

The message-flow aspects can be emphasized by using an explicit stereotype “messageFlow” for the transfer of messages between subsystems. This approach, illustrated in Figure 9-2, contrasts with the more abstract approach illustrated in Figure 9-1.

Note that the use of a transition between two ObjectFlowStates is not normal activity-graph usage, but is not prohibited by the UML semantics definitions.

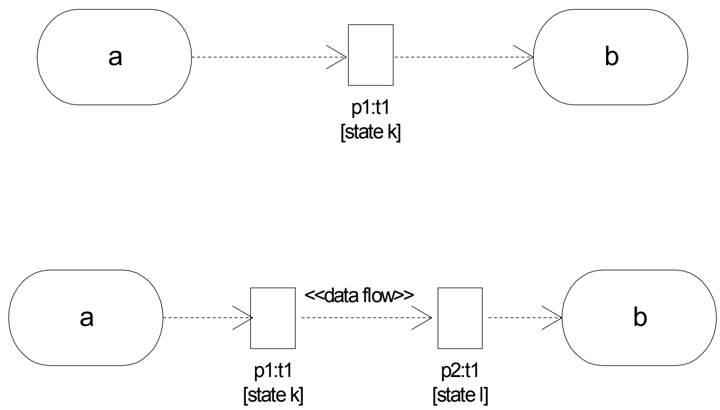



*Figure 9-2* Application of the "messageFlow" stereotype to emphasize data-flow aspects

Figure 9-3 illustrates the impact of applying this technique to the exchange example.

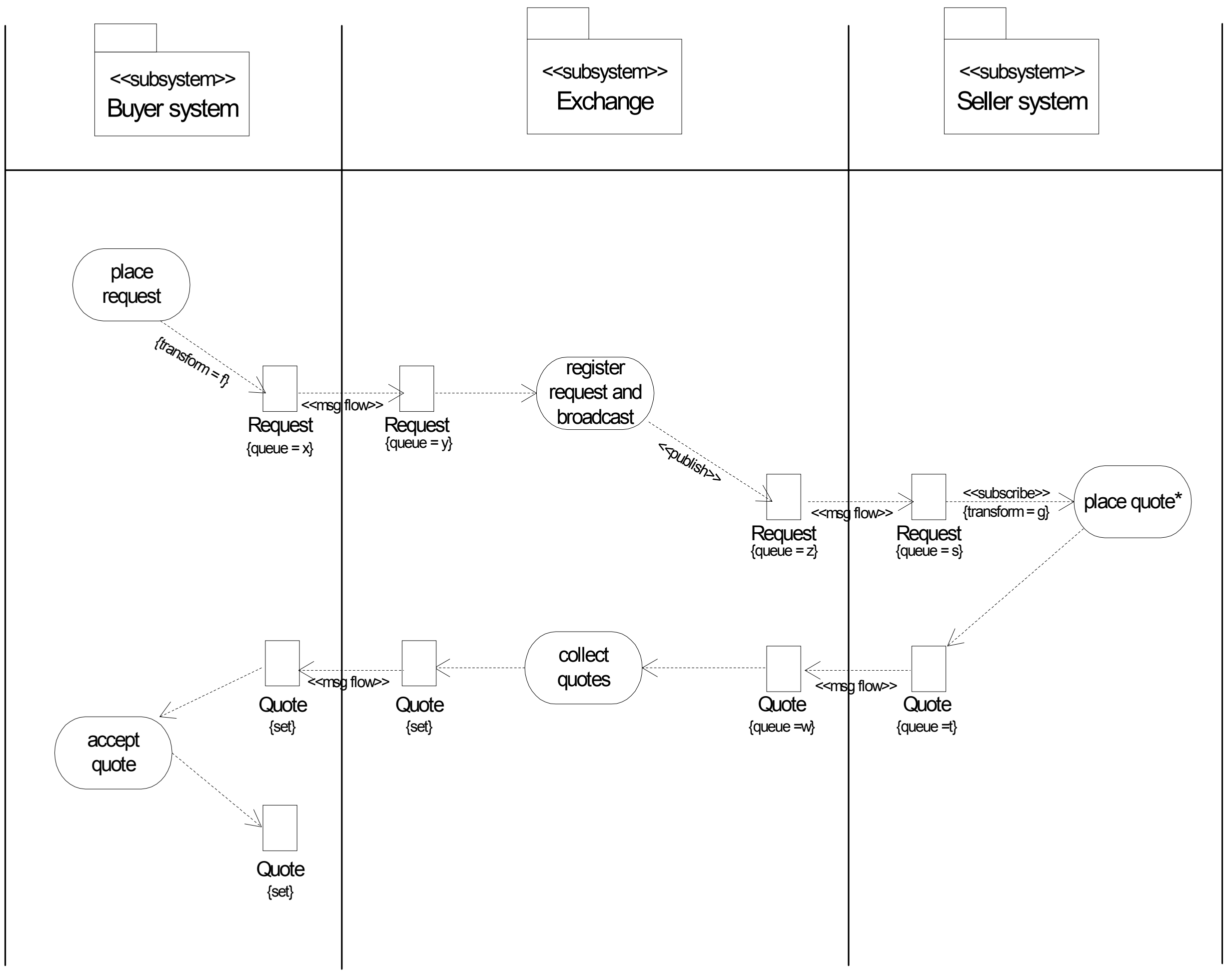



*Figure 9-3* Application integration example with "messageflow" stereotype (partial)

In this activity graph, the partitions represent the enterprise systems that require integration. Inside each partition, action states (i.e., activities) represent the invocation of application APIs. A transition between an activity and an object-flow state represents the production or consumption of a message. Transitions with the stereotype “messageFlow” represent message transfers across system boundaries. Message flows may be “point-to-point.” They may also be designated as “multicast” (according to a publish/subscribe protocol) by adding the stereotypes “publish” and “subscribe” to appropriate transitions.

### 9.2.3 Modeling control flow

In addition to message flow aspects, control flow aspects can be added to the process definition. In Figure 9-4, control flow transitions have been added within each of the component systems in a fragment of the Exchange example.

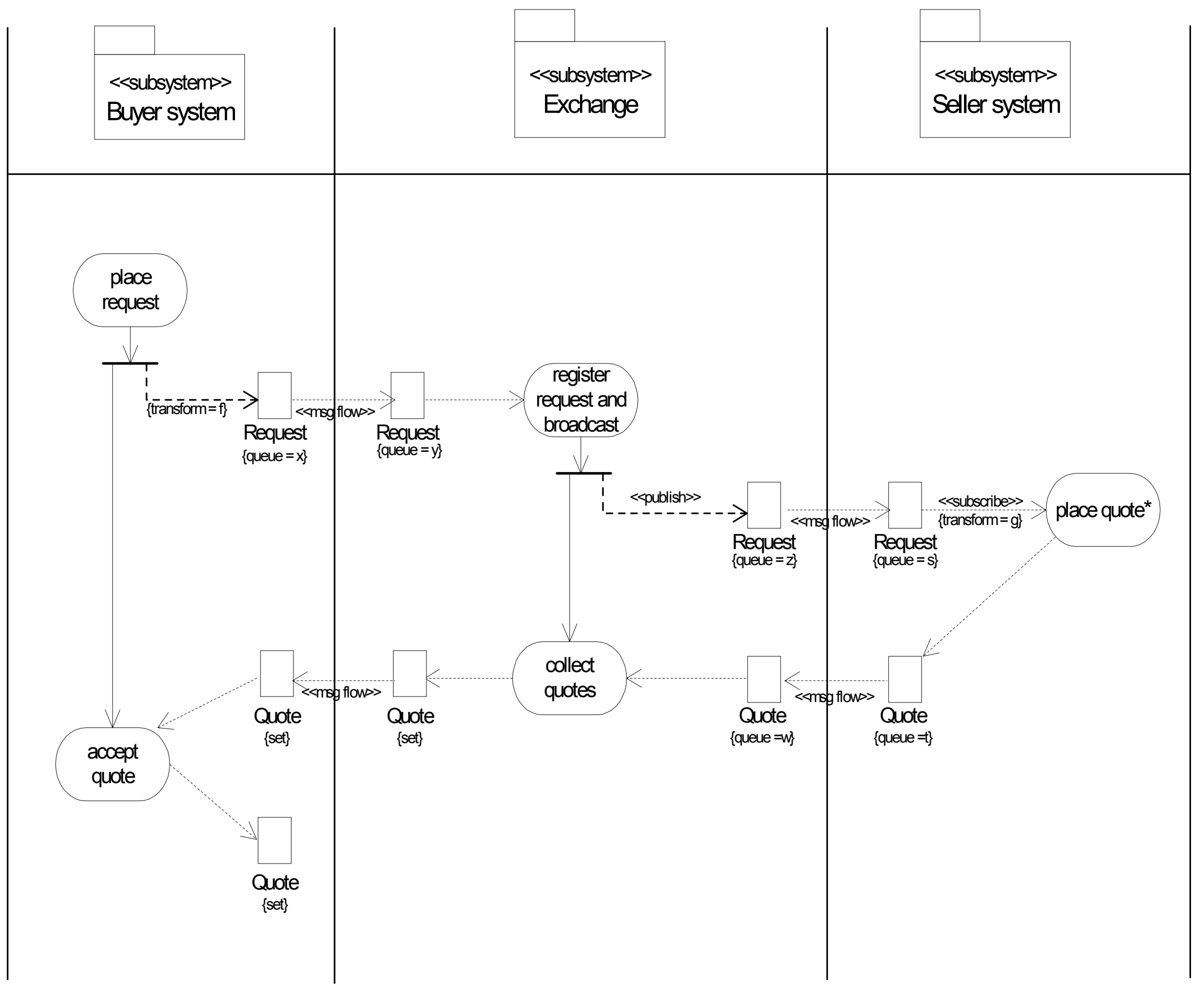



*Figure 9-4* Optional control flow transitions between activities within a single system

### 9.2.4 Abstracting detail by decomposition

Activities can be decomposed to show the constituent set of subactivities. An example of the decomposition of the integration step “Place Quote” is shown in Figure 9-5. In this step, an incoming Request message results in an outgoing Quote message. In the decomposition, a “connector” activity is responsible for handling the incoming message. Once a message is accepted by the system, a “transformer” activity transforms the message content to a locally acceptable format. Finally, “adapter” activities take this known input and adapt it into the legacy data store format. A subsequent "adapter" is responsible for invoking the legacy system. After the legacy application has run, a similar set of steps produces the outgoing message.

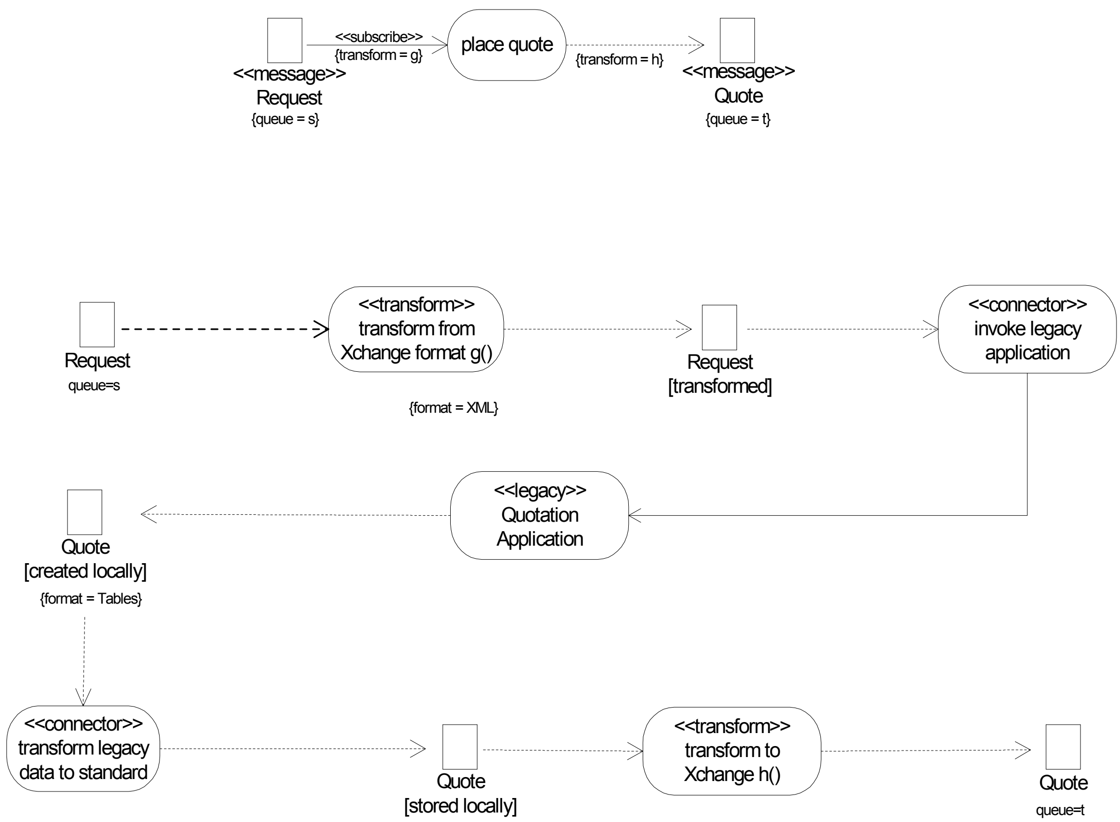



*Figure 9-5* Decomposition of the integration step "Place Quote" in the context of the Exchange example

It should be noted that the above figure constitutes a prototypical example - many variants of these will exist using additional operator activities (e.g., involving "router" and "filter" operators) and with varying process structure.

## *9.2.5 Further Fragmentary Examples*

Other activity graph constructions that can be used in modeling system-integration processes are described in the following subsections.

### *9.2.5.1 Multiple synchronized inputs and outputs*

Multiple synchronized inputs and/or outputs can be modeled with join and fork pseudo-states (see Figure 9-6).

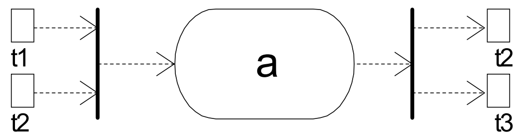



*Figure 9-6* Modeling multiple inputs and outputs with join and fork pseudo-states

### 9.2.5.2 *Internal dataflows within a subsystem*

An internal dataflow between two activities within a single system can be modeled with ObjectFlowStates (see Figure 9-7).

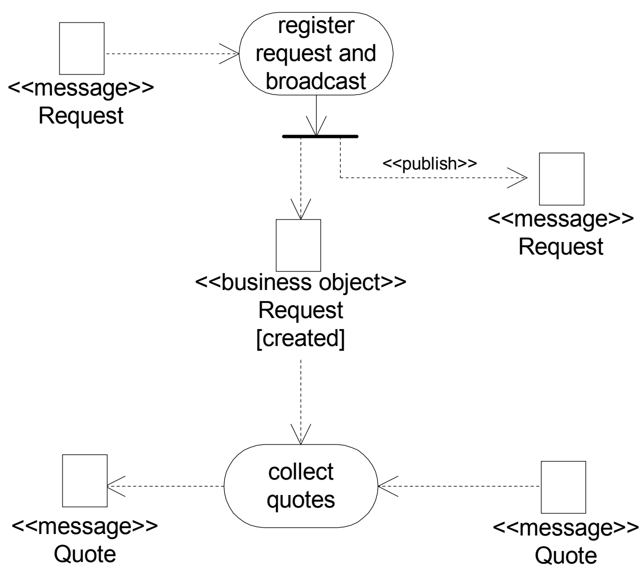



*Figure 9-7* Modeling internal data flow with object flow states

### 9.2.5.3 *Modeling decisions explicitly*

Decisions can be modeled with guards, either implicitly with multiple outgoing or explicitly by using a decision PseudoState - the latter approach is relevant for modeling content based routing where the middleware is responsible for rule execution (as opposed to embedded rules executed by applications) (see Figure 9-8).

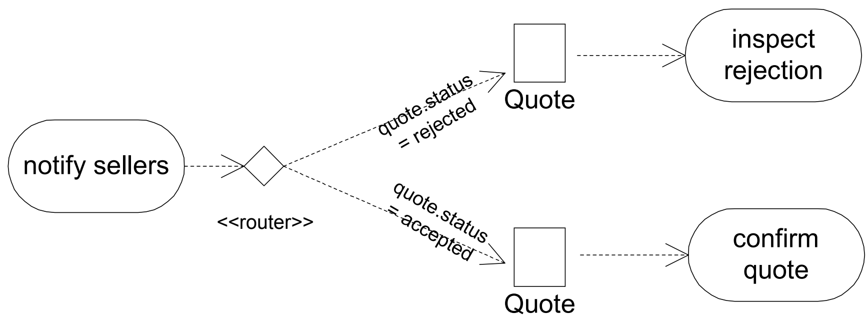



*Figure 9-8* Example of a decision node to model rule-based routing

### *9.2.5.4 Synchronization*

Synchronization is made explicit with Fork and Join PseudoStates - for instance, this can be used to model multiple parallel invocation of legacy systems in the case that there is more than one system (or more than one function) needing to be invoked (see Figure 9-9).

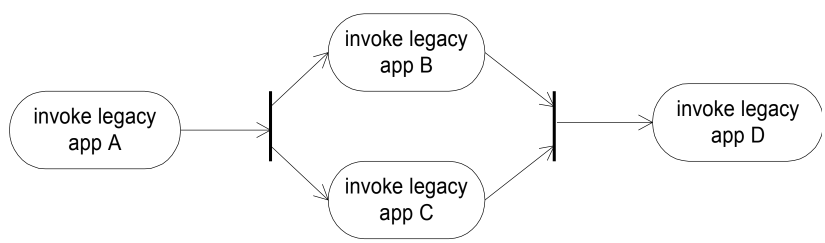



*Figure 9-9* Synchronization with forks and joins

### *9.2.5.5 Multiple concurrent invocations of activities*

Dynamic concurrent invocation of activities (where the number of actual activities invoked is determined at run time depending on the input) is denoted by a "*" symbol in the activity (see Figure 9-10).


*Figure 9-10* Dynamic concurrent invocation of an activity

### 9.2.5.6 Modeling business events explicitly

Events (as based on the definition of a signal in UML) can be added to transitions or they can be modeled explicitly as object-flow states. In the latter case, the underlying classifier is a signal, with attributes representing the event parameters. This can be useful for modeling "adapter" implementations that respond to events (e.g., a database trigger) and for systems that natively expose a required integration event on their interface (see Figure 9-11).

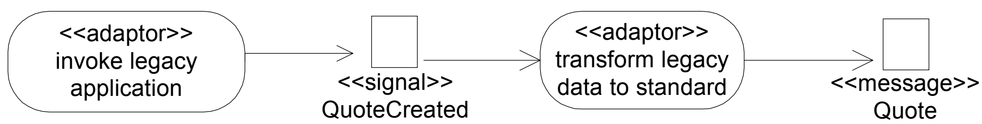



*Figure 9-11* Explicit modeling of an event for an adapter implementation

Integration processes will usually not be defined beyond this level of detail in activity graphs. The design of the interactions between the classes involved is best described using collaboration modeling, as discussed in *Collaboration Modeling* Chapter 8.

## 9.3 Profile Element Summary

The following is a summary of the activity-graph stereotypes and tagged values for modeling processes in the context of EAI. These stereotypes are in addition to the ones defined in Chapter 4. Tagged values on the activity stereotypes enable the linking of activities to their realization in terms of the Operator and Message Classifiers that implement the messaging functionality at the Collaboration level.

It should be noted that some stereotypes apply to more than one UML metaclass. In some cases, the metaclass name is given in brackets to indicate that this is a secondary modeling option. For instance, a "transform" stereotype is primarily attached to an activity, indicating that the activity is realized by a transform operator. A "transform" stereotype can also be attached to a transition as a secondary option that can be useful for models that are (to be) decomposed.

This profile definition assumes the UML 1.3 extension mechanism, which is string-based. In UML 1.4, references to metaclasses can be used as an alternative to name based strings. Furthermore, in UML 1.4, multiple stereotypes can be applied to an element.

### 9.3.1 Stereotypes

Table 9-1 defines the basic stereotypes. Tagged values for these stereotypes are defined in Table 9-2.

*Table 9-1* Behavior stereotypes for modeling EAI system-integration processes

| Stereotype | UML metaclass | Comments / constraints |
|---|---|---|
| "integration process | "ActivityGraph | A system integration process in the context of EAI. |
| "message | "ObjectFlowState | A data element that is interchanged between two systems. The ObjectFlowState “inherits” the stereotype from the Signal classifier with stereotype “message” that it points to, if one is defined at this stage (tagged value defined below). The underlying Classifier of the ObjectFlowState represents the “content” of the message (to be added as a Signal parameter during design). The production or consumption of a message by an activity is modeled with a “flow” Transition. |
| "flow | "Transition | A flow is an exchange of data between two systems. An abstract stereotype. A “flow” may optionally have an associated guard condition. |
| "messageflow | "Transition | A “messageflow” is a subtype of a “flow” where the Transition is to or from a “message.” Abbreviated to “msg flow.” The production or consumption of a message constitutes an event in EAI context. Note that general business events are modeled as Signals in UML. |
| "connector | "ActionState, ActivityGraph(Transition) | A “connector” is a simple or compound activity that converts a specific kind of message from some system-specific format into a specified message-content type, or vice versa. |
| "operator | "ActionState, ActivityGraph(Transition) | An “operator” is an activity that acts upon messages as they flow between systems. Note: if the activity has more than one message as input or output, then the operator must be a Compound Operator. |
| "transform | "ActionState, ActivityGraph,(Transition) | A kind of operator that transforms datasets from one format to another. An instantiable subtype stereotype of “operator.” |
| "filter | "ActionState, ActivityGraph,(Transition) | A kind of operator that filters messages according to a rule. |
| "router | "ActionState, ActivityGraph(PseudoState) | A kind of operator that determines an outgoing channel based on a rule. |
| "stream | "ActionState, ActivityGraph(Transition) | A kind of operator. |
| "adapter | "ActionState, ActivityGraph(Transition) | A kind of operator, indicating a wrapper activity that encapsulates dataflow and / or controlflow to and from a legacy system, e.g., an operator that performs invocation and associated marshaling. |
| "publish | "Transition(ActionState) | A kind of operator, indicating that there is a publisher - subscriber protocol involved in the message transmission (default is point - to - point). |
| "subscribe | "Transition(ActionState) | A kind of operator, indicating that there is a publisher - subscriber protocol involved in the message transmission (default is point - to - point). |
| "legacy | "ActionState, ActivityGraph(PseudoState) | A “legacy” is any existing application that participates in an integration. |

## 9.3.2 Tagged Values

Table 9-2 defines the extended “meta-properties” (i.e., tagged values) for the stereotypes defined above, and a number of general supporting tags. Some of these tags are references to classes and signals that are defined using the modeling framework defined in Chapter 4. These references define the realization of the messaging functionality specified in the integration process (e.g., using sequence diagrams).

*Table 9-2* Tagged values with the stereotypes defined in the previous table

| **Tagged value** | **UML Metaclass / stereotype** | **Notes** |
|---|---|---|
| signalImplementation : String | "message" ObjectFlowState | Indicates that an ObjectFlowState with stereotype “message” is realized by a Signal (note: in UML 1.4 this becomes a reference to a Signal Classifier stereotyped “message” instead of a string). Note the base classifier reference of the ObjectFlowState points to its content class. |
| sourceImplementation : String | "message" ObjectFlowState (ActionState,ActivityGraph) | Indicates that an ObjectFlowState with stereotype “message” is realized as a Source Classifier at a detailed level (note: in UML 1.4 this becomes a reference to a Classifier instead of a string). Alternatively, when applied to an activity it indicates that in the detailed realization this activity has an associated Source Classifier. Optional property. |
| targetImplementation : String | "message" ObjectFlowState (ActionState,ActivityGraph) | Indicates that an ObjectFlowState with stereotype “message” is realized as a Target Classifier at a detailed level (note: in UML 1.4 this becomes a reference to a Classifier instead of a string). Alternatively, when applied to an activity it indicates that in the detailed realization this activity has an associated Target Classifier. Optional property. |
| queueName : String | "message" ObjectFlowState(Transition) | Indicates the name of the queue to be used (note: in UML 1.4 this becomes a reference to a Queue Classifier instead of a string). |
| queueProtocol : String {JMS, IBM MQ, Oracle AQ, ...} | "message" ObjectFlowState (Transition) | Indicates the target implementation type for the queue. |
| format : String {XML, ...} | "message" ObjectFlowState (Transition) | Indicates the target implementation format for the message. |
| isSet : Boolean | "message" ObjectFlowState | Indicates that an ObjectFlowState contains a set of messages. Shorthand notation for a type expression, e.g., “Set of Quote.” |
| communicationProtocol : String {http, iiop, smtp, ...} | Transition | Indicates the target communication protocol to be used. |

*Table 9-2* Tagged values with the stereotypes defined in the previous table

| Tagged value | UML Metaclass / stereotype | Notes |
|---|---|---|
| operatorImplementation : String | "operator" Activity (Transition) | Indicates a reference to the Classifier that realizes the activity. Note: in the case of a subtype of “operator” such as “transform,” “connector,” “publish,” or “adapter” this tagged value indicates the transformationImplementation Classifier, connectorImplementation Classifier, etc. (note: in UML 1.4 this becomes a reference to a Classifier with stereotype “operator” instead of a string). |
| operationImplementation : String | "adapter" Activity (Transition) | Indicates a reference to a public operation of a “legacy” system (note: in UML 1.4 this becomes a reference to an Operation on a Classifier instead of a string). |
| directoryNameEntry : String | Subsystem | Indicates the implementation target name for the subsystem. |

## *9.3.3 Mapping to EAI Metamodel*

Although the activity graph elements are largely used at a higher level of abstraction, as illustrated above, it is possible to decompose some aspects of the model sufficiently to map directly onto the same metaobjects as the collaboration model. Table 9-3 lists these mappings and identifies the appropriate metaclasses to map the other activity graph profile elements.

*Table 9-3* Mapping from Activity Graph Stereotypes to EAI Metaclasses

| Stereotype | EAI metaclass | Comments/Constraints |
|---|---|---|
| "integration process | "FCMComposition | The “integration process” is the overall context for the model. It is merely the aggregation of all the elements of the activity graph. |
| "message | "EAIMessageContent | Direct mapping. |
| "flow | "FCMLink | Not being constrained to only connecting terminals, a “flow” in the activity graph profile, which is a stereotype on the UML metaclass Transition, is more generic than EAILink. |
| "messageflow | "FCMDataLink | A “messageflow” is an example of a Transition that does not directly connect Terminals. It represents the propagation of a message from one system to another, probably implemented as queue-to-queue propagation, but at this level of abstraction it is not appropriate to specify that. |
| "connector | "EAIPrimitiveOperator | Direct mapping |
| "operator | "EAIPrimitiveOperator | Direct mapping |
| "transform | "EAITransformer | Direct mapping |
| "filter | "EAIFilter | Direct mapping |
| "router | "EAIRouter | Direct mapping |
| "stream | "EAIStream | Direct mapping |

*Table 9-3* Mapping from Activity Graph Stereotypes to EAI Metaclasses

| **Stereotype** | **EAI metaclass** | **Comments/Constraints** |
|---|---|---|
| "adapter | "EAISourceAdapter/EAITargetAdapter | Whether an instance of an activity model “adapter” is an instance of an EAISourceAdapter or an EAITargetAdapter can be inferred from the context. |
| "publish | "FCMLink | The application of this stereotype is specifying a constraint on the underlying queue implementation, but the link itself is not the queue. This is an example of where the analysis model contains a design hint but is not of itself a design specification. |
| "subscribe | "FCMLink | See “publish.” |
| "legacy | "FCMNode | “legacy” is a necessary component of the activity profile because it provides a reference point for the business, but in the integration itself “legacy” has no behavior, so it is mapped to the generic FCMNode. |

# *Part 4 - Proof of Concept*

## *Contents*

This section contains the following chapters.



This section provides a proof of concept for the profile by giving examples of the use of the profile for actual EAI modeling. An example is provided that is relevant to both of the scenarios of the Scope that are covered by this specification and uses collaboration modeling. In Chapter 11, a variant of part of this example is presented in the CCA of the UML Profile for EDOC.

# Example: Connectivity and Information Sharing 10

This chapter shows how the UML Profile for EAI can be used to model the integration of applications for a brokerage firm using collaboration modeling. The chapter is structured as follows:

- Section 10.1 provides a brief description of what a brokerage firm does. This provides some explanation of the domain in which the models are being developed.
- The following sections describe aspects of the brokerage firm's systems, which are then captured in models expressed using the collaboration profile of Chapter 8.

## 10.1 The Brokerage Business

A brokerage firm accepts orders for stock trades from various parties:

- Direct from customers
- From partner brokerages in other countries
- From investment managers

The job of the brokerage firm is, essentially, to enact the trades requested in those orders and then send notifications back to the customer.

The focus of the modeling in this chapter is the handling of orders from partner brokerages and from investment managers. This requires an architecture integrating with the systems used by these stakeholders, which allows order events from systems external to the enterprise to be transformed into a common format and then filtered, elaborated, and processed. Notifications need to be generated and sent back to the originating systems.

The overall architecture for this integration is depicted in Figure 10-1.

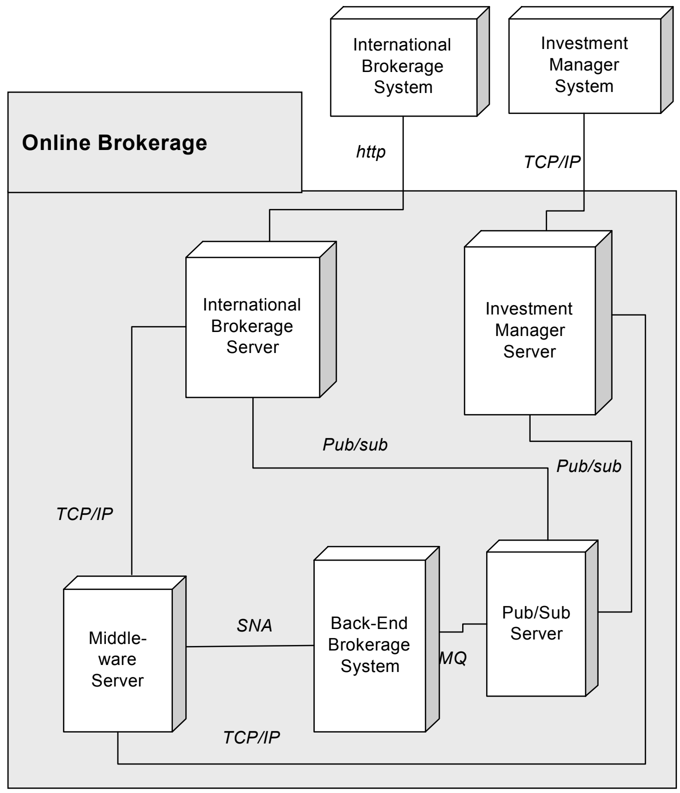



*Figure 10-1* As-is architecture for international and investment managers

International customers (i.e., customers outside the U.S.) are served by a brokerage system in their own country. This system keeps track of portfolios for its customers. If those customers wish to trade U.S. securities, those trade requests are serviced by the on-line brokerage system.

Investment managers manage portfolios on behalf of customers with large or complex holdings. They use the brokerage system to place trades and to get information about various securities for their customers. Different investment-manager firms use different software for portfolio management.

Using the UML profile, we elaborate a model of the architecture of this system.

## 10.2 Connection of Enterprises to the Online Brokerage System

The on-line brokerage system is connected to five external systems. This is shown in the collaboration diagram in Figure 10-2.

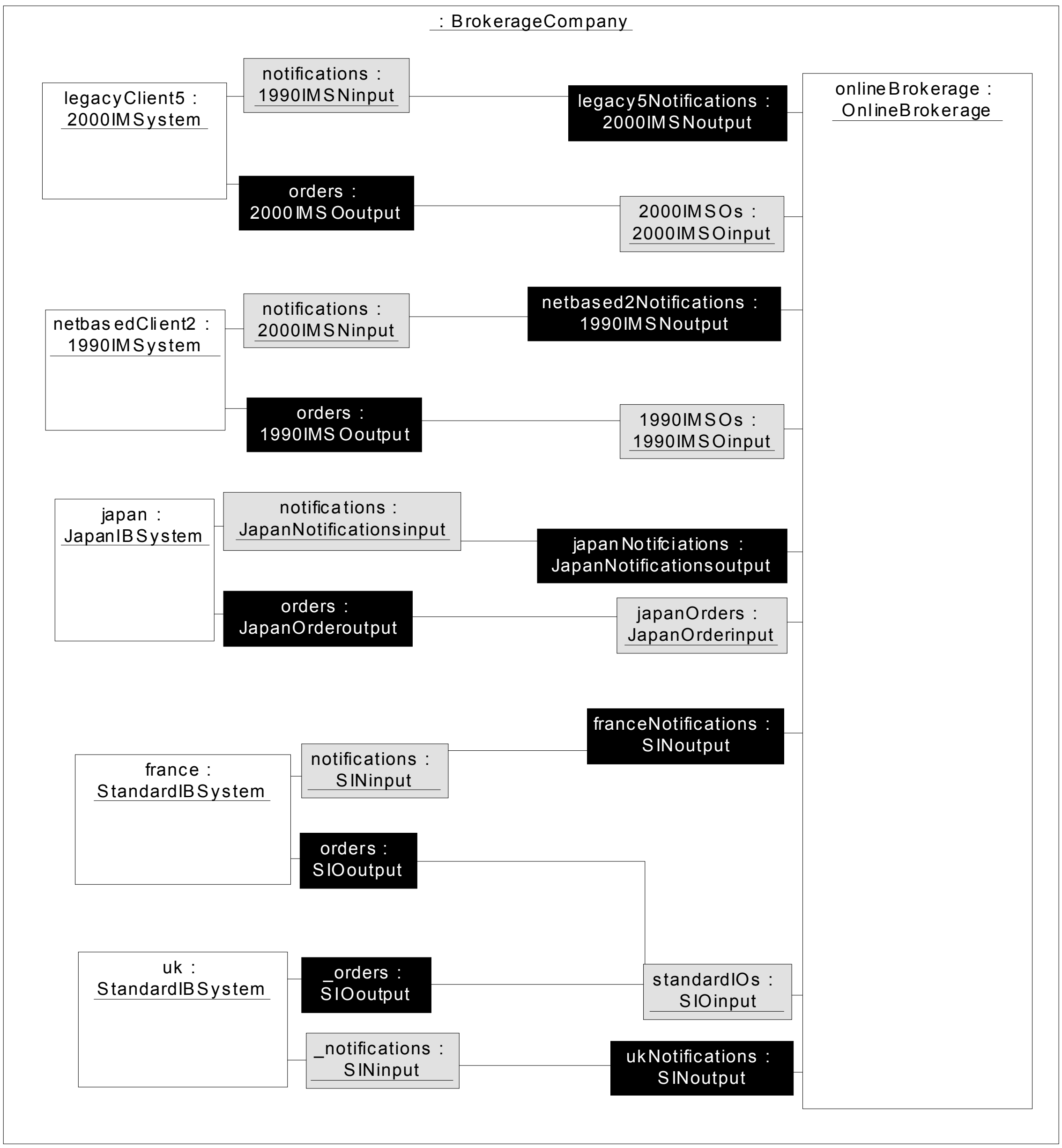



*Figure 10-2* Brokerage company - component connections

This diagram highlights the two key processes involved:

1. The processing of orders entered into the system
2. The publication of notifications about processed orders

Thus each system external to the online brokerage has an output terminal for issuing orders to be sent on to the online brokerage system and an input terminal for receiving notifications back. Interestingly, although input streams of orders in the same format may be merged (e.g., the output terminals of uk and france both connect to the same input terminal of the on-line brokerage), the output streams of notifications will not. There are good business reasons (such as confidentiality) to ensure, for example, that only notifications for France go to France and not also to the UK.

It has been left unspecified as to whether the connections between external systems and the on-line brokerage are synchronous or asynchronous, although they are likely to be asynchronous.

Figure 10-3 is the corresponding class diagram, which declares the components of the brokerage-company operator, where primitive operators are used to model the systems external to the online brokerage. Notice that two of the external systems (uk and france) are of the same type. The components of the compound operator representing the on-line brokerage system will be explored in the subsequent sections.

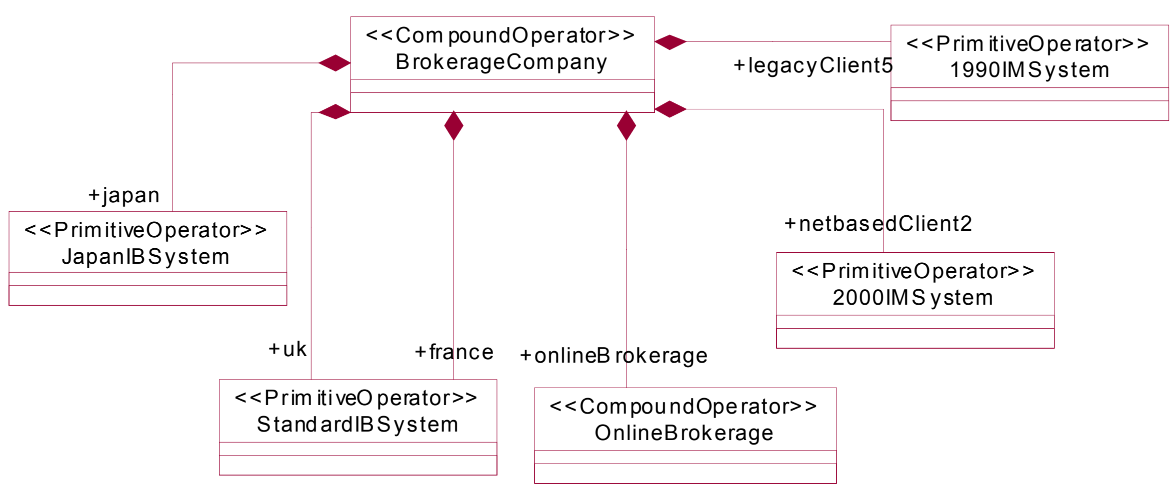



*Figure 10-3* Brokerage company - components

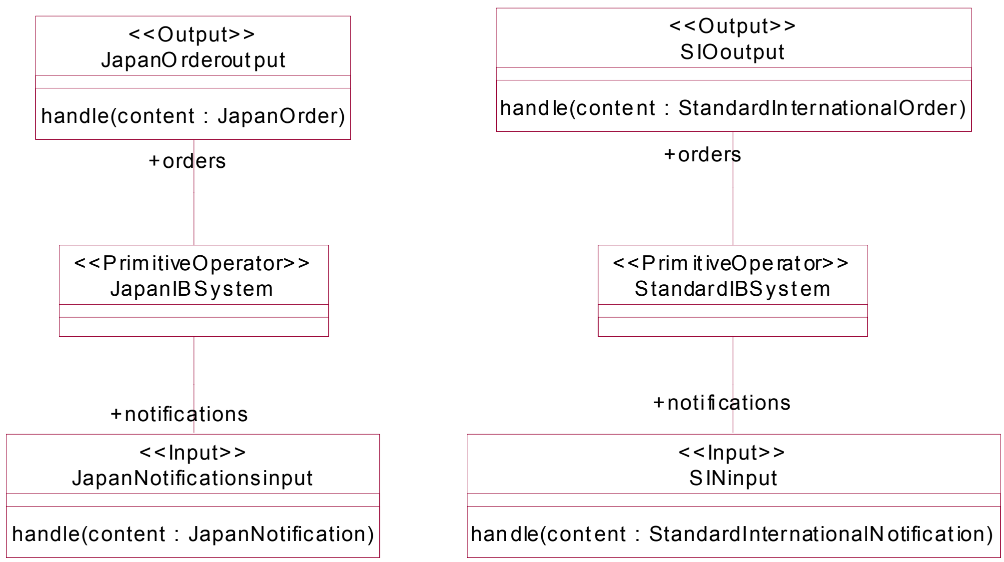



*Figure 10-4* International brokerage systems - terminals

Figure 10-4 to Figure 10-6 define the terminals of all these operators and of the on-line brokerage. The on-line brokerage must handle four different formats of orders and notifications. Two of the systems (france and uk) use the same formats.

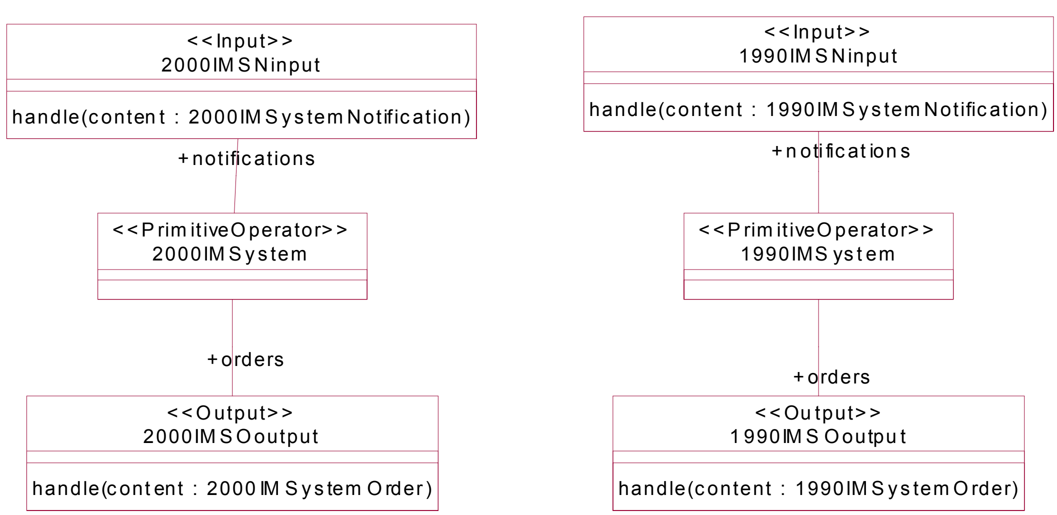



*Figure 10-5* Investment-manager systems - terminals


*Figure 10-6* On-line brokerage system - terminals

## *10.3 The On-line Brokerage System*

The on-line brokerage system is a compound of an international brokerage server, an investment manager server, a middleware server, a back-end brokerage system, and a Pub/Sub server. The components are declared in Figure 10-7, and the way in which they are connected together is specified in Figure 10-8.

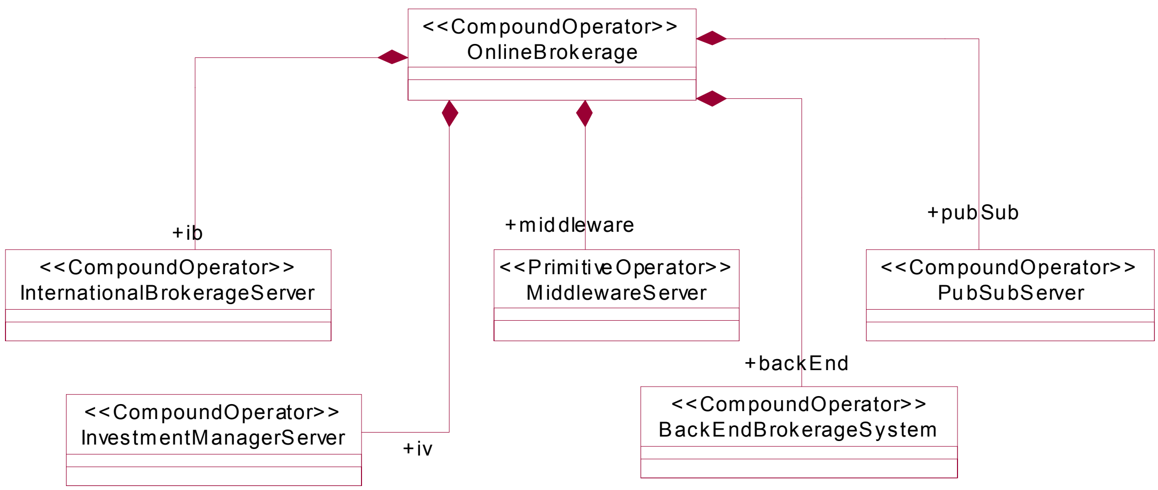



*Figure 10-7* On-line brokerage system - components

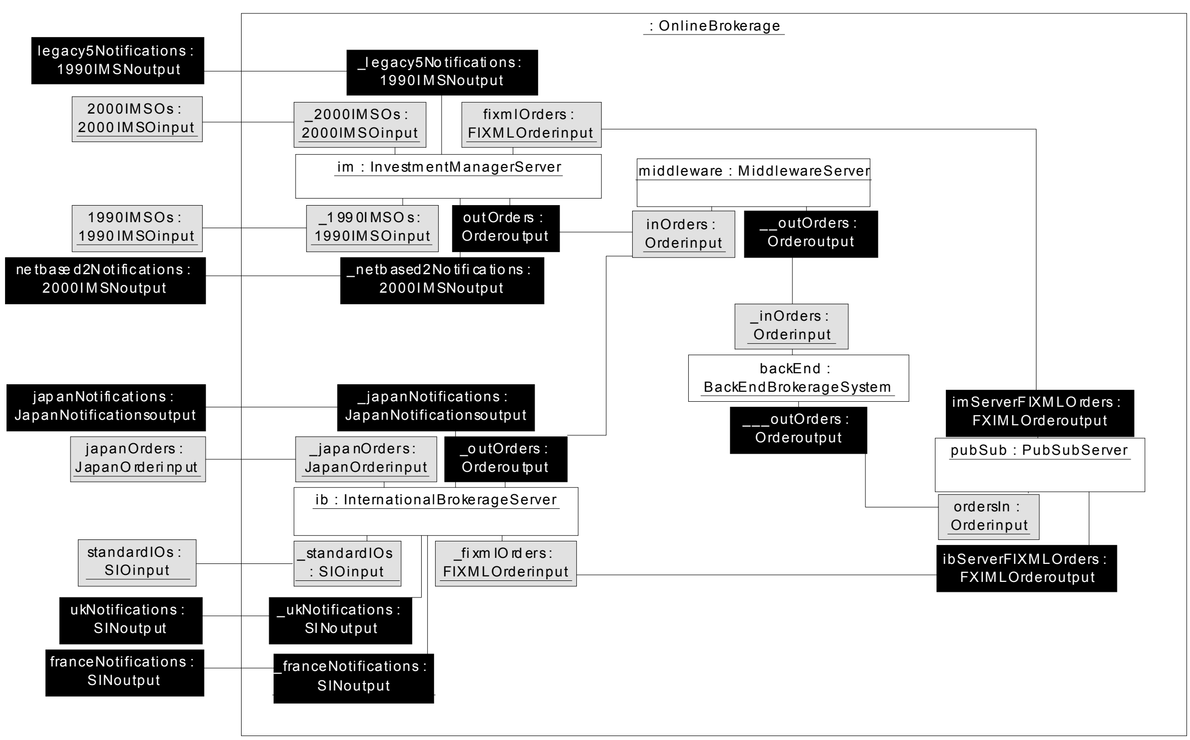



*Figure 10-8* On-line brokerage - component connections

Orders from international brokers are handled by the international-brokerage server, and orders from the investment managers are handled by the investment-manager server. These systems convert the orders into a common format and pass them on to the middleware server, which forwards them to the back-end brokerage server. There the orders are processed, and ownership information is added. On exit from this system they are passed to the Pub/Sub server, which routes the processed orders back to the IB or IM system, depending on which one generated the order. The IB and IM systems generate notifications from the processed orders, which are passed on to the external systems as appropriate.

The terminals for each of these systems are defined by Figure 10-9 to Figure 10-13. As usual, these diagrams give details about the formats of message handled by the terminals of each system.

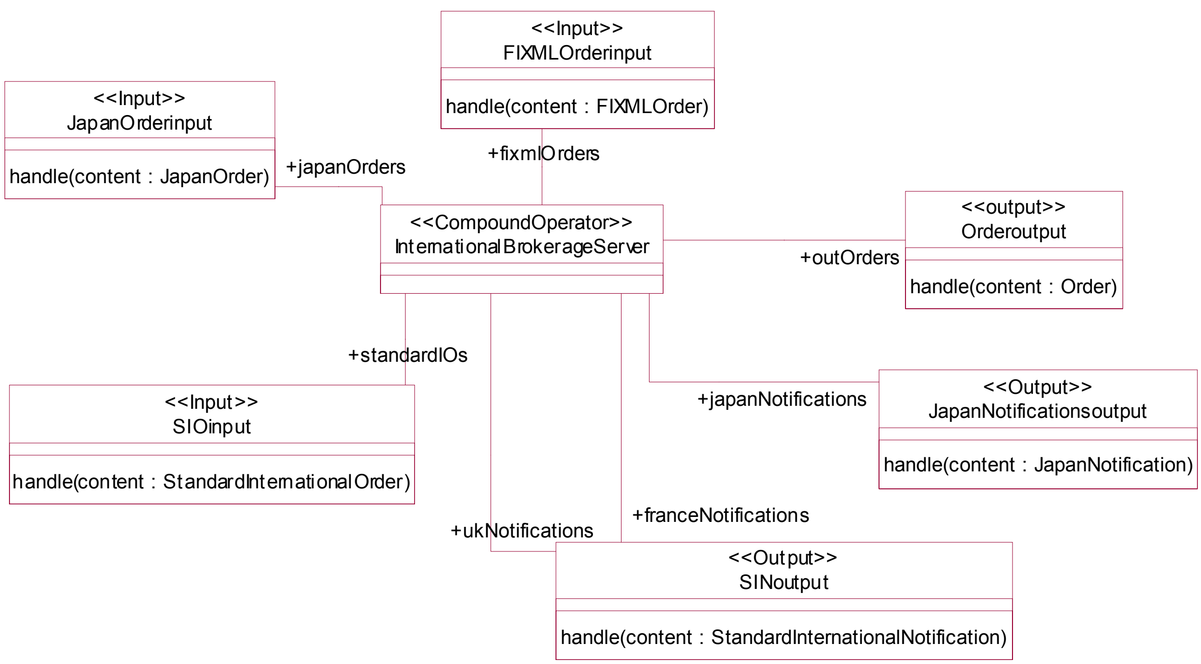



*Figure 10-9* International brokerage server - terminals

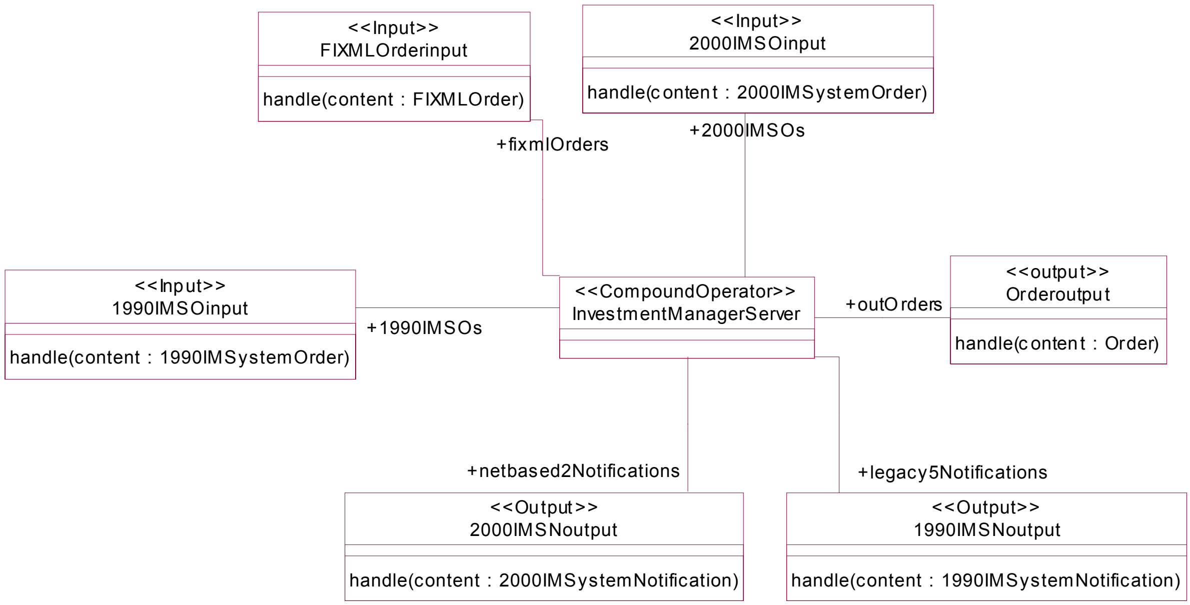



*Figure 10-10* Investment-manager server - terminals

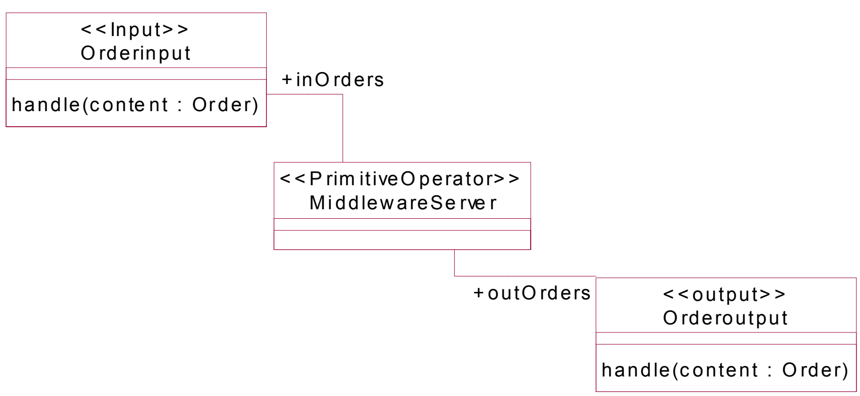



*Figure 10-11* Middleware server - terminals

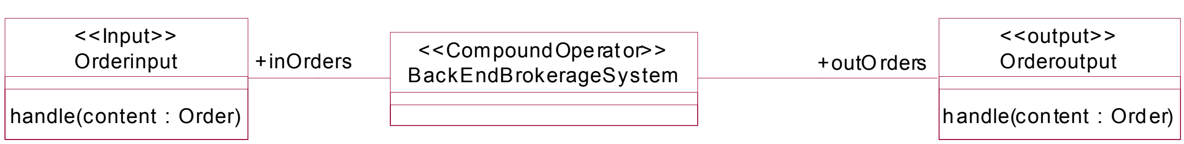



*Figure 10-12* Back-end brokerage system - terminals

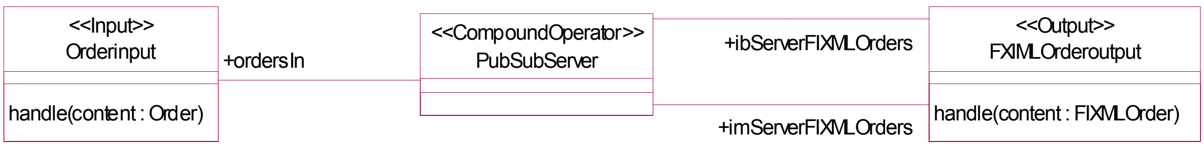



*Figure 10-13* Pub/sub server - terminals

We are now ready to examine the workings of each of the components of the online brokerage server.

## 10.4 International Brokerage Server

### 10.4.1 Orders

For international customers, order flow is as follows:

- When a customer of an international broker places an order for execution of a trade involving securities traded on a U.S. exchange, the order is forwarded to the online brokerage for execution, which then passes on the order to its international brokerage server.
- The international brokerage server transforms the order into the standard format understood by the back-end systems.

### 10.4.2 Notifications

The International server will send notifications to the international broker in near real time. These are generated from the order events received from the Pub/Sub server.

The diagrams defining the components of the international-brokerage server (IBS) are given by Figure 10-14 and Figure 10-15.

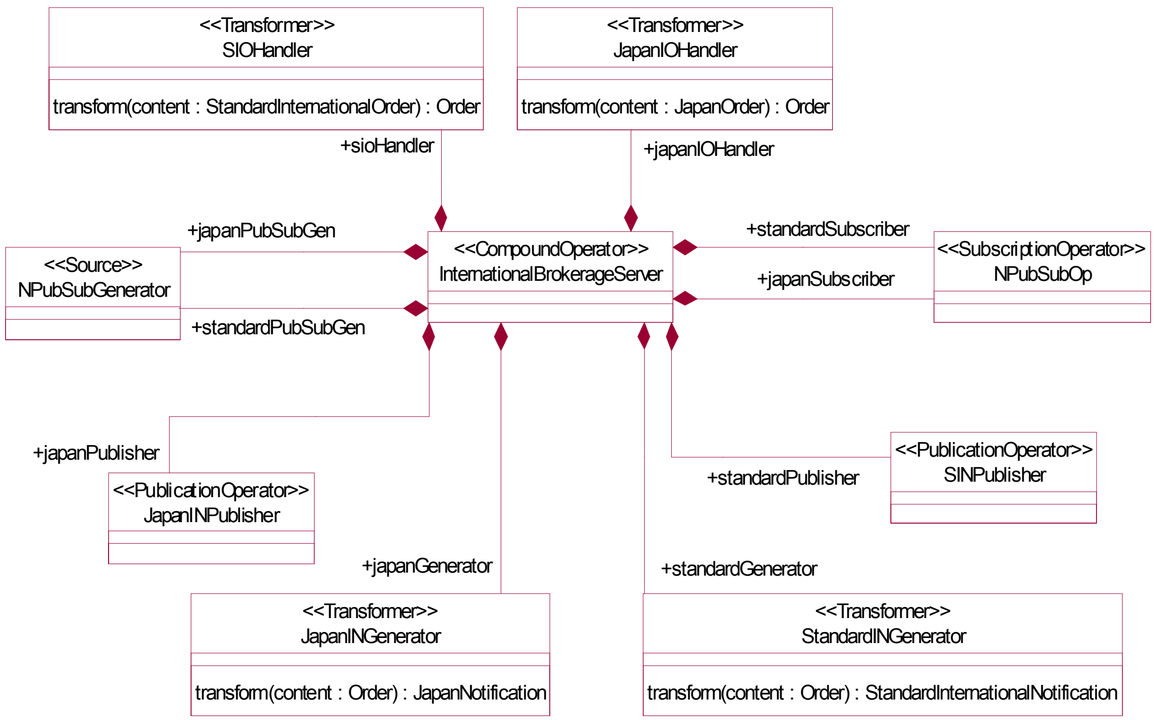



*Figure 10-14* IBS - components

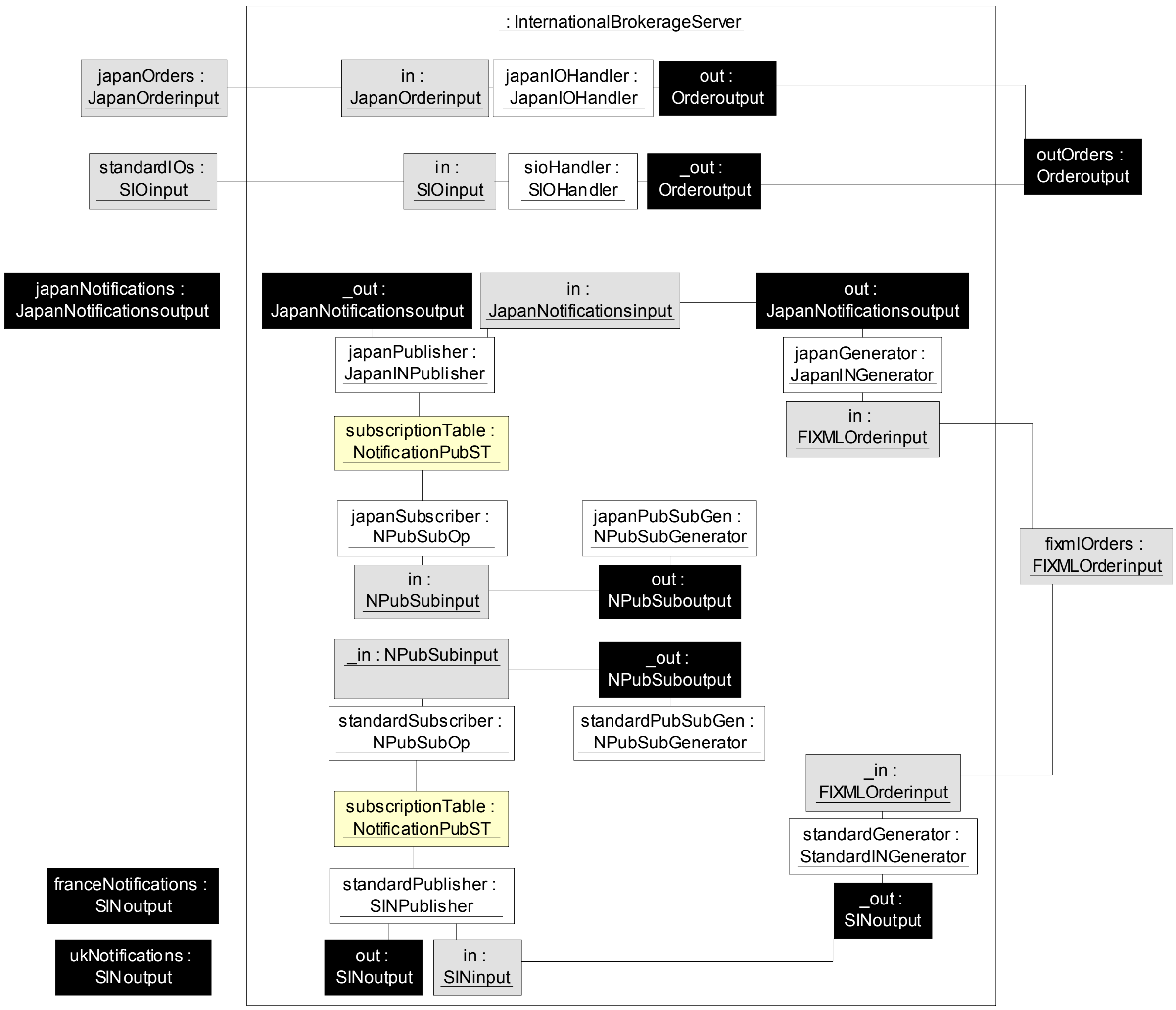



*Figure 10-15* IBS - component connections

For orders, there needs to be one transformer per input format, which converts that input format to the standard format.

For notifications, there needs to be one transformer per notification format. As there may be many external systems that handle the same format (in this case uk and france work with the same format), and these are likely to come and go, it makes sense to use a dynamically configurable publication operator on the output of each transformer. This avoids having a separate transformer for each system; a transformer is only needed for each format. This means, in turn, that connection of a new system to

Pub/Sub will only require a notification output terminal to be set up for connection to the notification input terminal of that system. New internal components will not be required.

The publication operator will dynamically connect to the appropriate notification output terminals on a message-by-message basis, as dictated by its subscription table. This explains why the notification output terminals of the IBS are not connected to any of its components. For each publisher, a combination of a subscription operator and source is used to generate subscriptions from some underlying system.

The definitions of the terminals of the components have been omitted, as they are relatively straightforward.

## 10.5 Investment Manager Server

### 10.5.1 Orders

Things are a little bit different for order placement from the Investment Manager systems:

- First of all, these systems utilize different tools for placing orders. So the Investment Manager server has to convert these different formats into a common format that can be handled by the middleware server and the back-end systems.
- Secondly, the investment managers commonly perform complex operations like balancing portfolios for a number of their customers at one time. This means sending a single message that can include multiple buy and sell orders for a single account and can include transactions on behalf of multiple accounts at the same time. It makes sense to think of all the transactions related to a single account as a unit of work in this context. The Investment Manager Server decomposes these complex messages and turns them into single order requests that are placed with the back-end systems.

Thus the handling of orders by the investment manager server is similar to that of the international brokerage server. The only difference is that the transformers generate batch orders in a standard format, and these then feed into a transformer, which takes a single batch order as input and generates multiple output messages in the standard order format.

### 10.5.2 Notifications

As with orders, the investment management server may batch up any number of notifications for transmission to its partners.

The modeling of the investment manager server, with respect to notifications, is similar to that of the international brokerage server. The only difference is that there must be an aggregator that generates batch orders from the order stream. They can then be fed on to the transformers and publishers.

## 10.6 Middleware Server and Back-End Brokerage System

Orders for international and investment customers go through the standard path for the brokerage system. They are routed to the middleware server, which forwards them to the back-end systems for execution. No additional modeling for the middleware server is required at this level.

The back-end brokerage system is responsible for processing the orders. As orders are processed and the order database is updated, this triggers events that mark changes in the state of the order to be published. At this point, the following things happen:

- The order is checked for “account ownership.” Accounts belong to different organizations within the enterprise. In particular, the order events are examined at this point to determine whether or not the account belongs to the international or to the investment manager system. To make the determination requires extracting information from the customer databases.
- A further filter is then checked based on the type of order event. Not all order events are published from this back-end system.
- If the filter is passed, then a transformation is made of a database record into a COBOL copybook format. The information about account ownership is added to the order event.

The processing of orders is modeled by a primitive operator, which here has been called orderProcessor and is of type BackEndProcessingSystem. The other three stages of order manipulation are modeled by two filters and a transformer. These are declared in Figure 10-16, and the way in which they are connected together is specified by Figure 10-17.

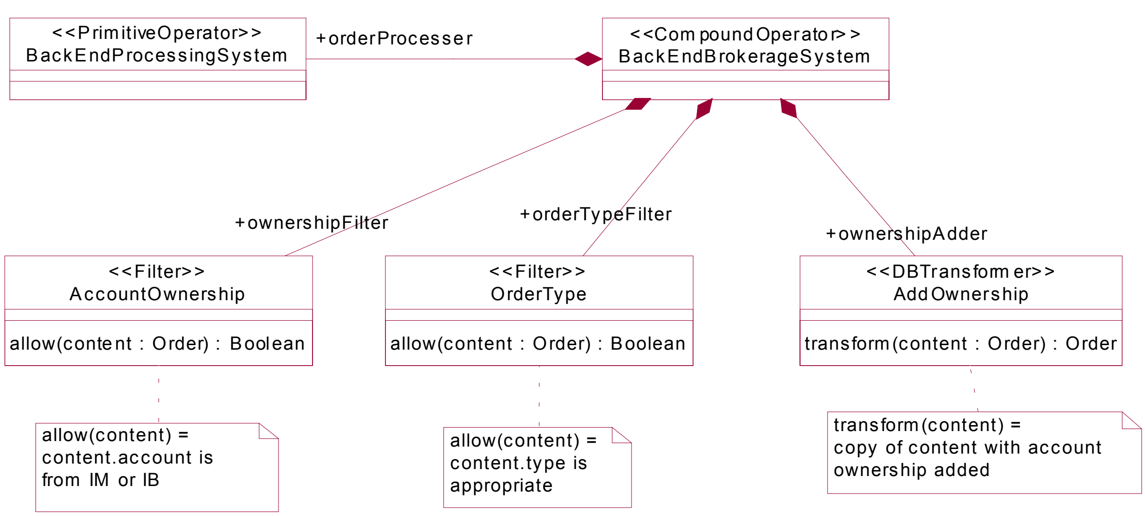



*Figure 10-16* Back-end brokerage system - components

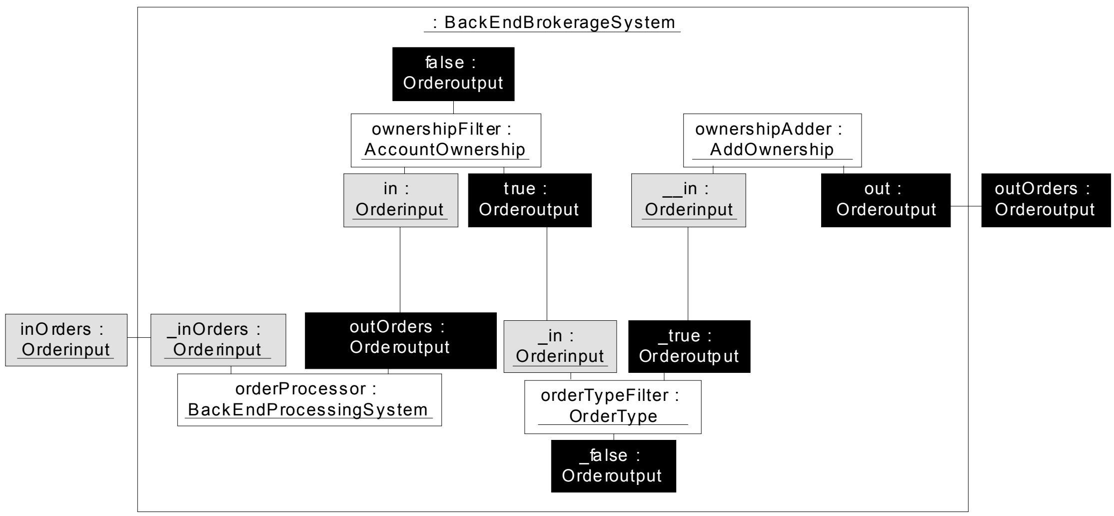



*Figure 10-17* Back-end brokerage system - component connections

The message-content format handled by the terminals of the filter and transformer can be deduced from the definition of the allow and transform operations, so we have omitted them here. The terminals for BackEndProcessingSystem are defined by Figure 10-18.

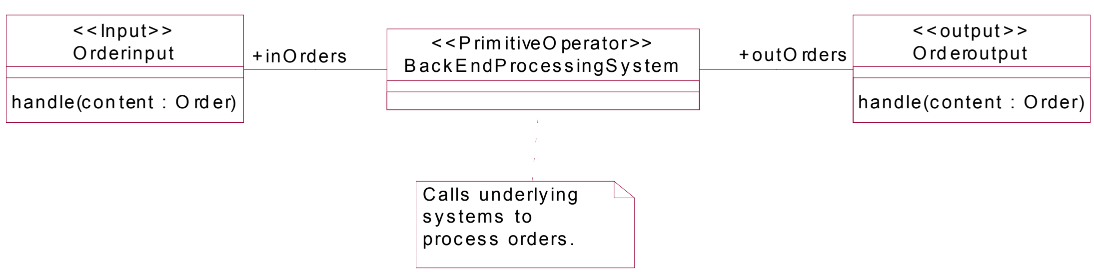



*Figure 10-18* Back-end processing system - terminals

## 10.7 Publication

The order event is then pushed to a Pub/Sub server. It accomplishes the following tasks:

- It transforms the order event into FIXML (a set of XML DTDs for the Financial Industry eXchange - FIX - protocol format).
- It publishes the event with a subject that includes the notion of ownership. The international and institutional customer servers subscribe to different order events. The international server subscribes to events that pertain to its customers, and the institutional server does likewise.

The Pub/Sub server can be modeled quite simply. The first point requires a transformer. Although the second point mentions publish and subscribe, dynamic subscription (a key part of the publication and subscription operators) is not required in this case. Rather, subscriptions are set up statically to filter messages based on their topic, and so this can be shown as a filter instead.

The definition of the components is given in Figure 10-19, and their configuration is given in Figure 10-20.

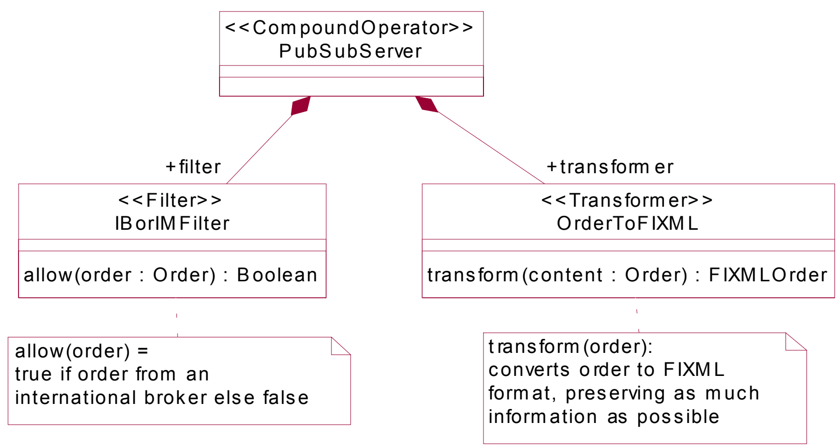



*Figure 10-19* Pub/sub server - components

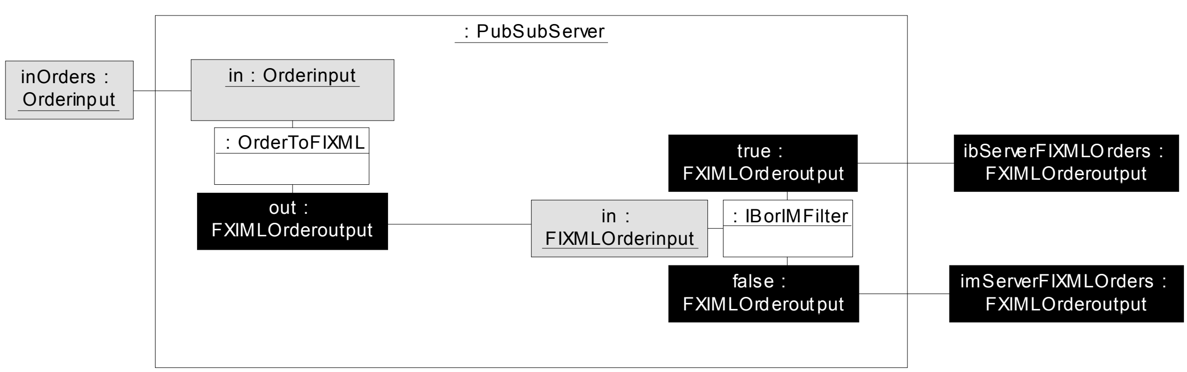



*Figure 10-20* Pub/sub server - component connections

The terminal specifications for the filter and the transformer have been omitted, as they can be deduced from the declaration of the allow and transform operations.

# Example Using the EDOC CCA 11

## 11.1 Example

The example in this section is based on a variant of that in Chapter 10. It illustrates the use of the Component Collaboration Architecture (CCA) of the UML Profile for EDOC. The high-level view in Figure 11-1 is a variant of that in Figure 10-2 on page 10-3.

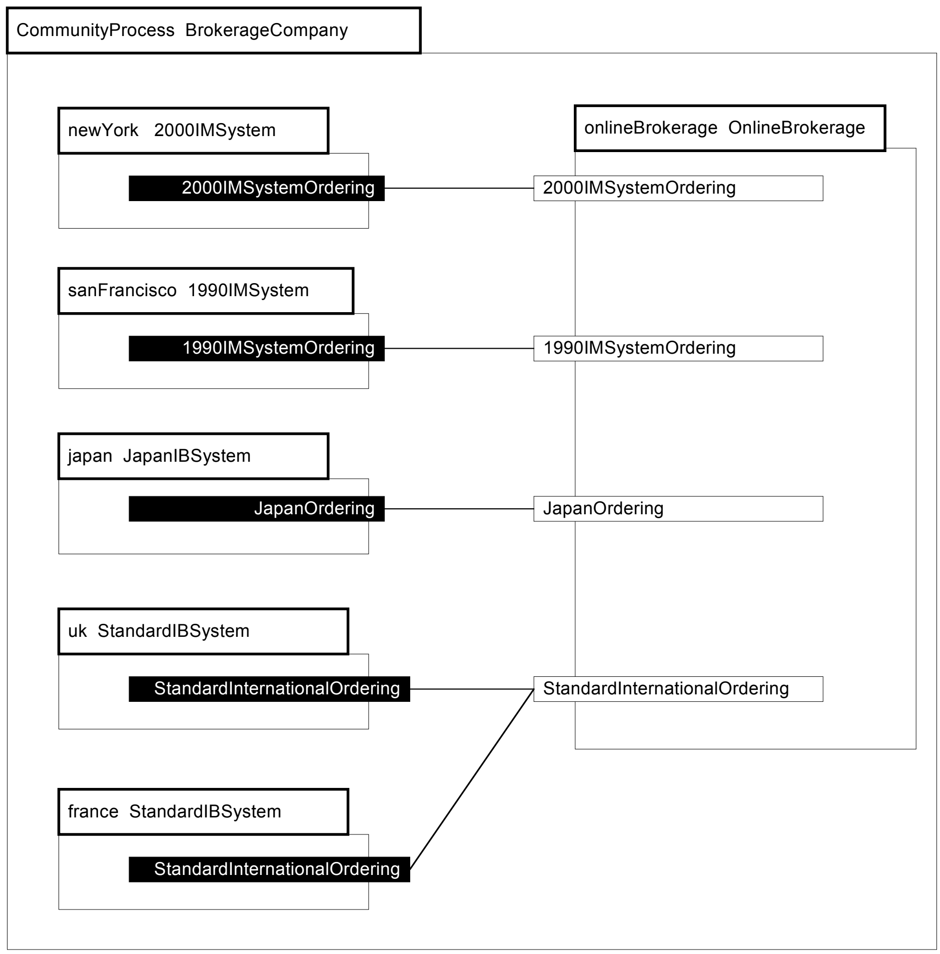



*Figure 11-1* BrokerageCompany component connections

The next two figures show the components.

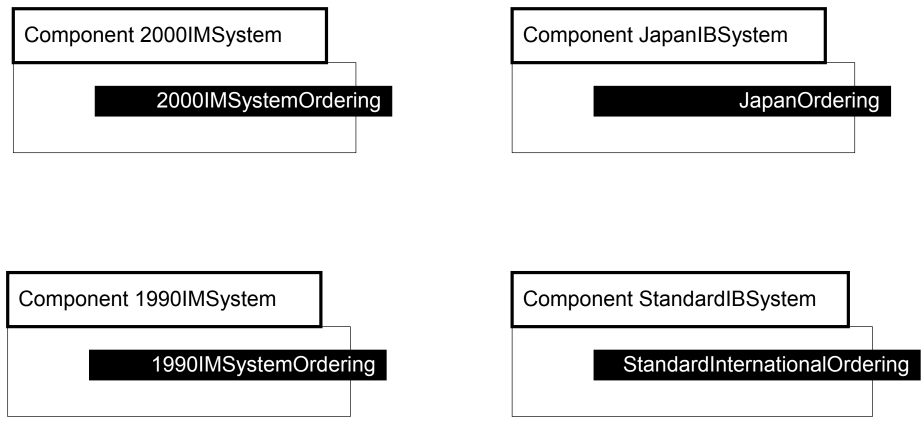



*Figure 11-2* Ordering Components

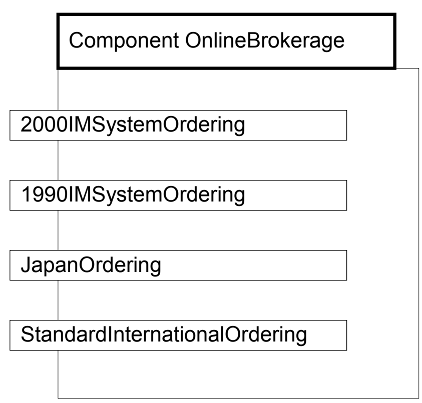



*Figure 11-3* OnlineBrokerage Component

The Protocols are in Figure 11-4 to Figure 11-7.

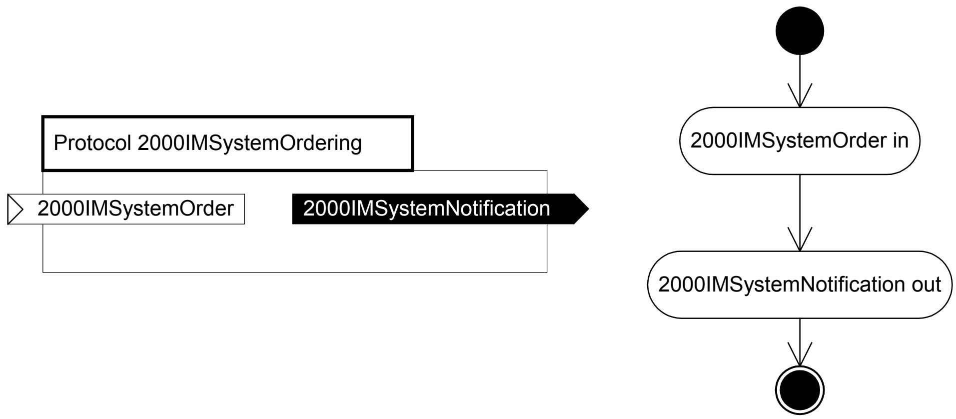



*Figure 11-4* 2000IMSystemOrdering Protocol

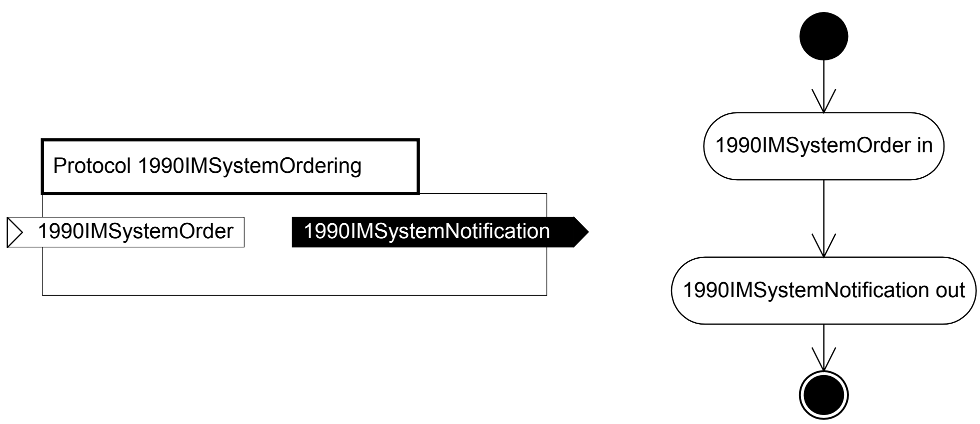



*Figure 11-5* 1999IMSystemOrdering Protocol

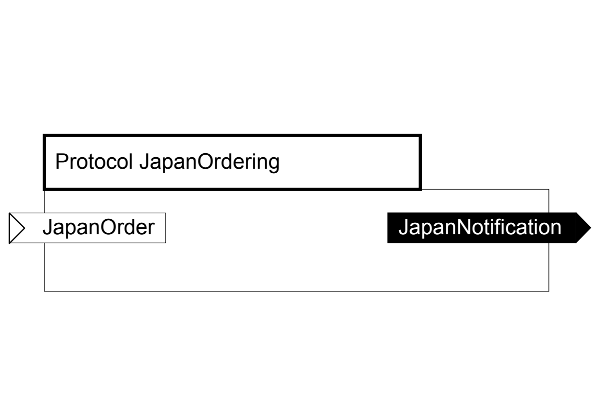



*Figure 11-6* JapanOrdering Protocol

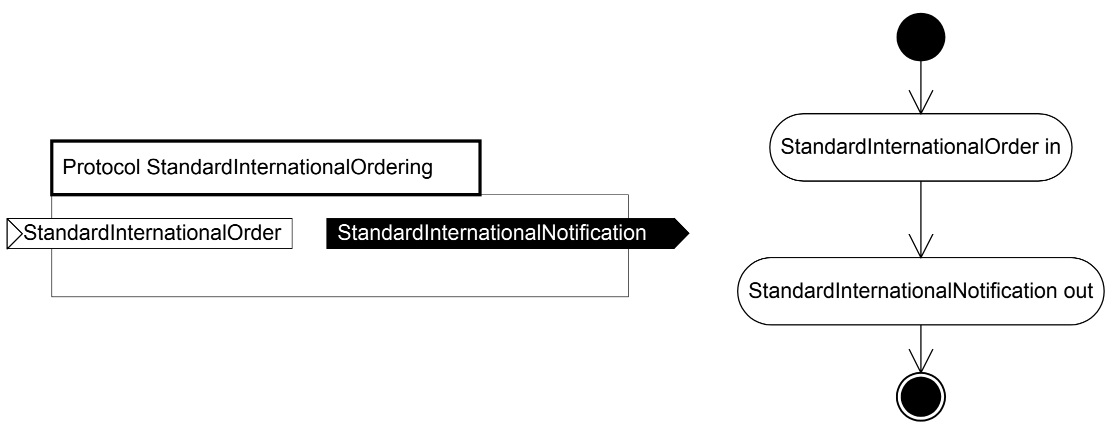



*Figure 11-7* StandardInternationalOrdering Protocol

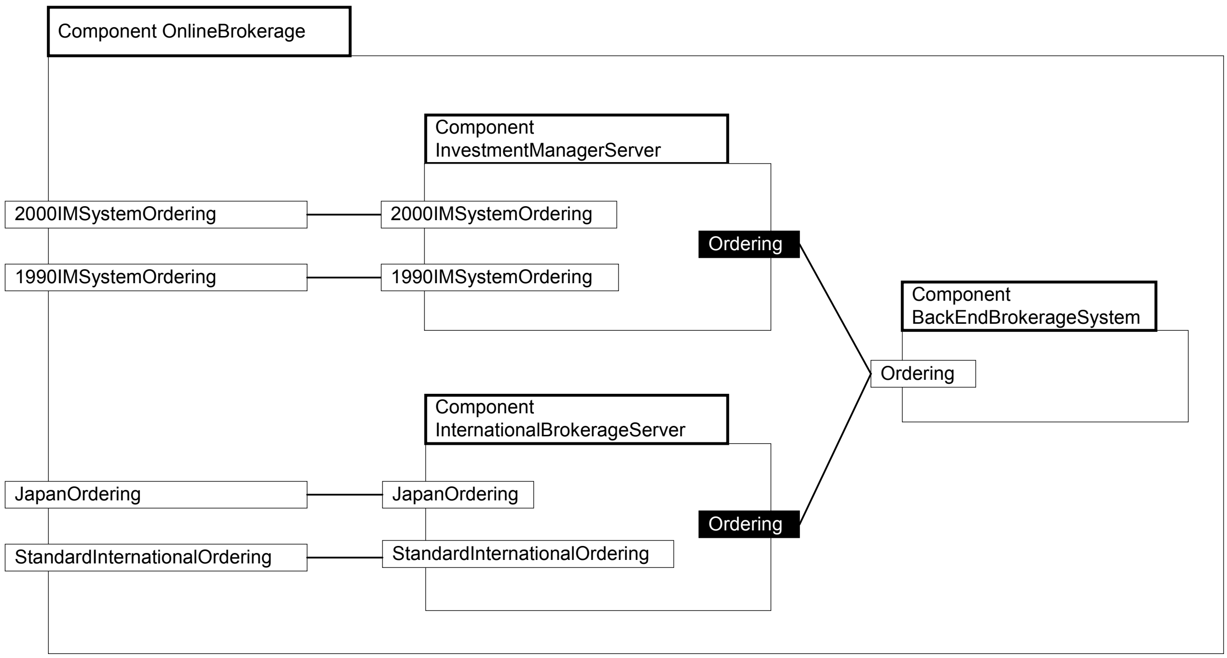



*Figure 11-8* Detail of OnlineBrokerage Component

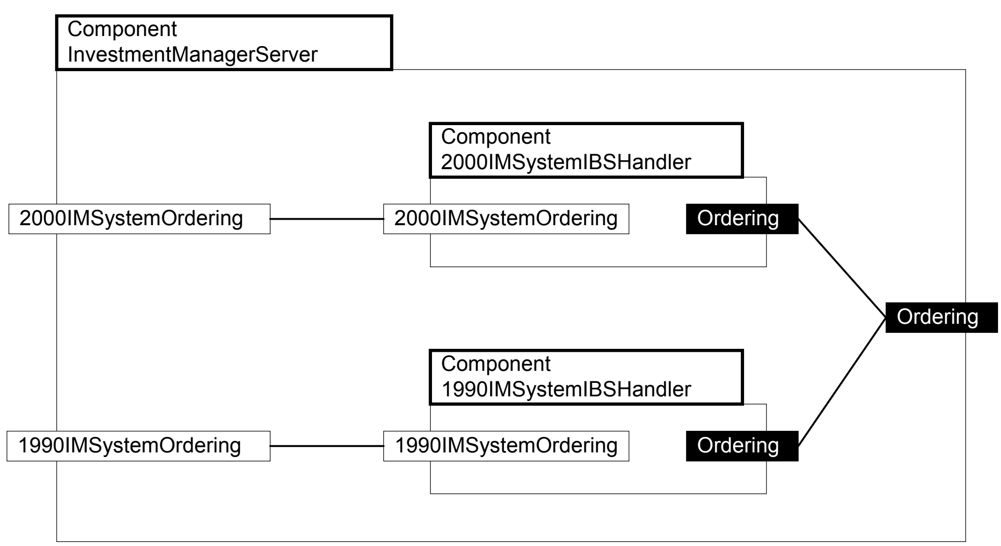



*Figure 11-9* Detail of InvestmentManagerServer Component

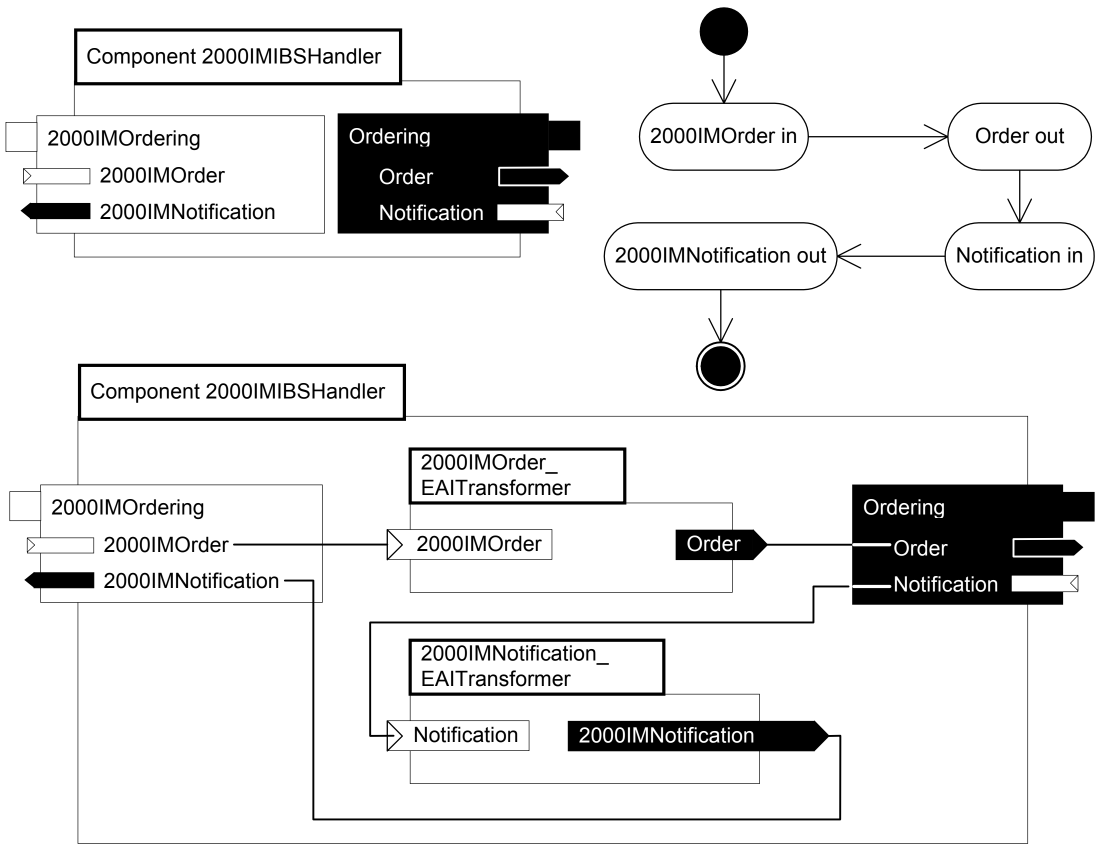



*Figure 11-10* Detail of 2000IMIBSHandler Component

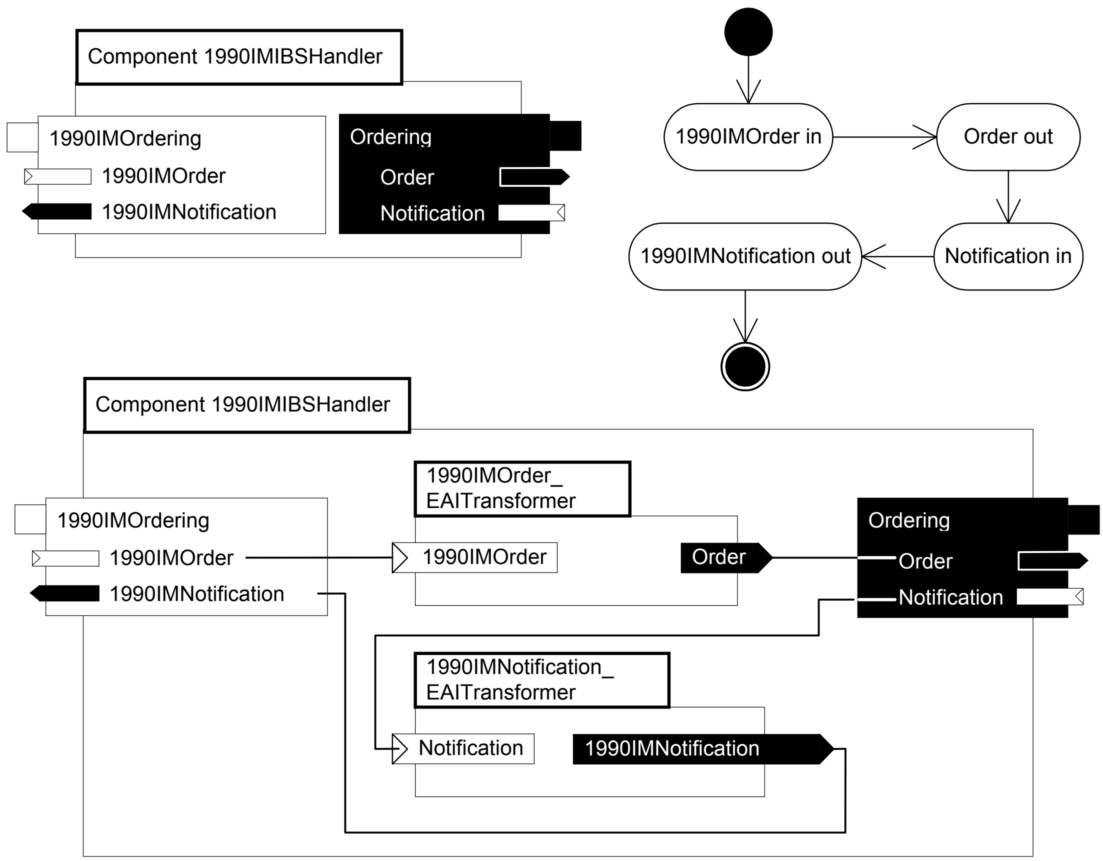



*Figure 11-11* Detail of 1990IMIBSHandler Component

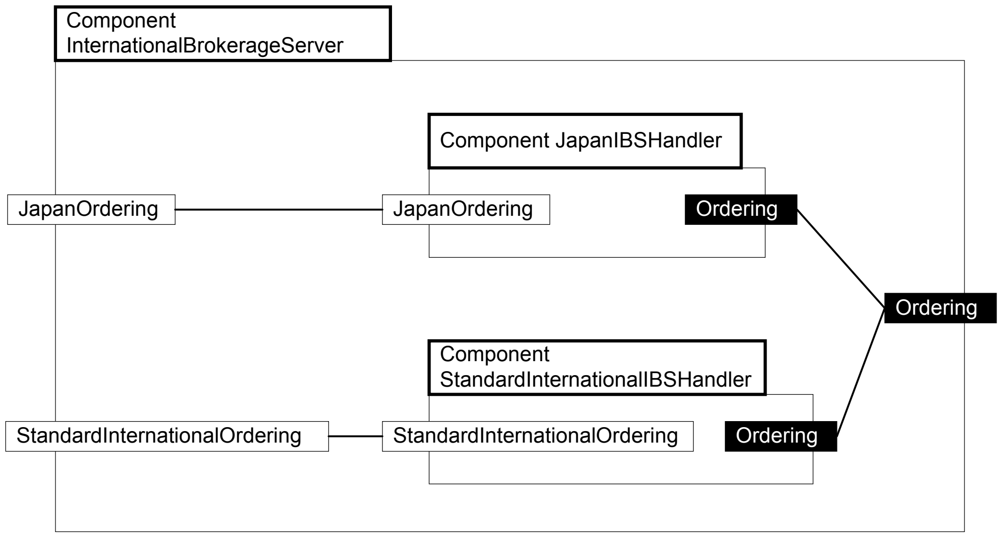



*Figure 11-12* Detail of InternationalBrokerageServer Component

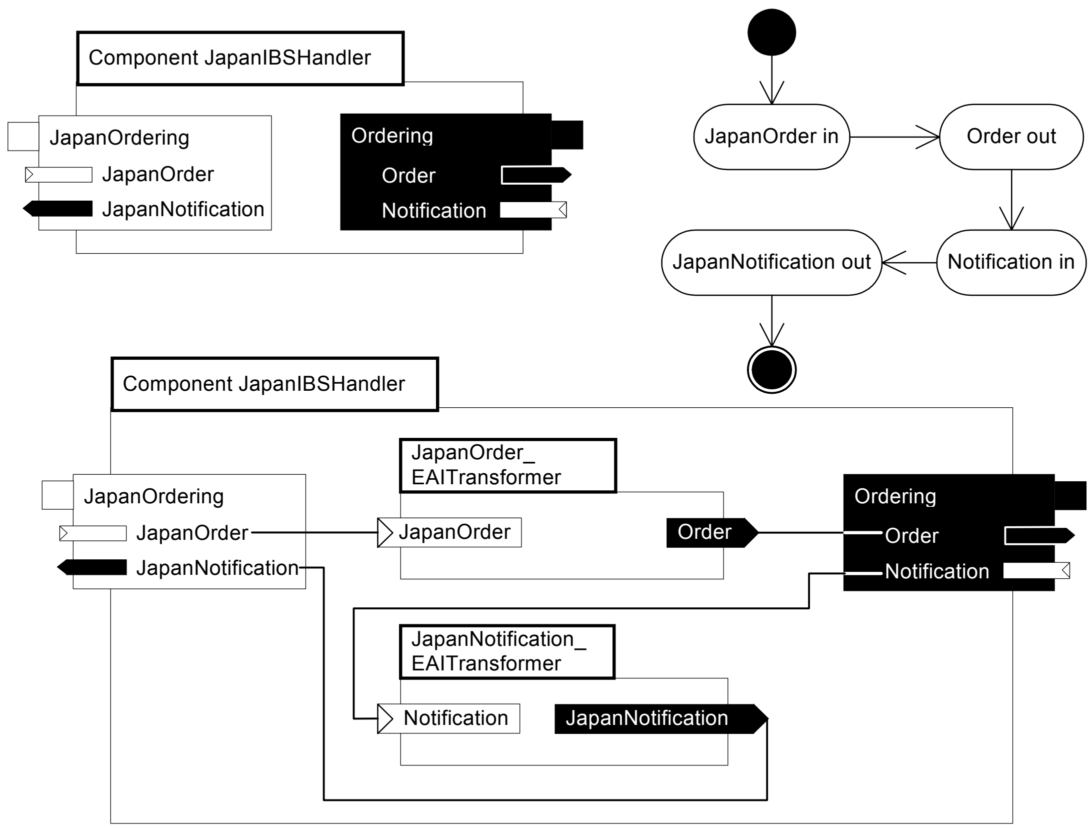



*Figure 11-13* Detail of JapanIMIBSHandler Component

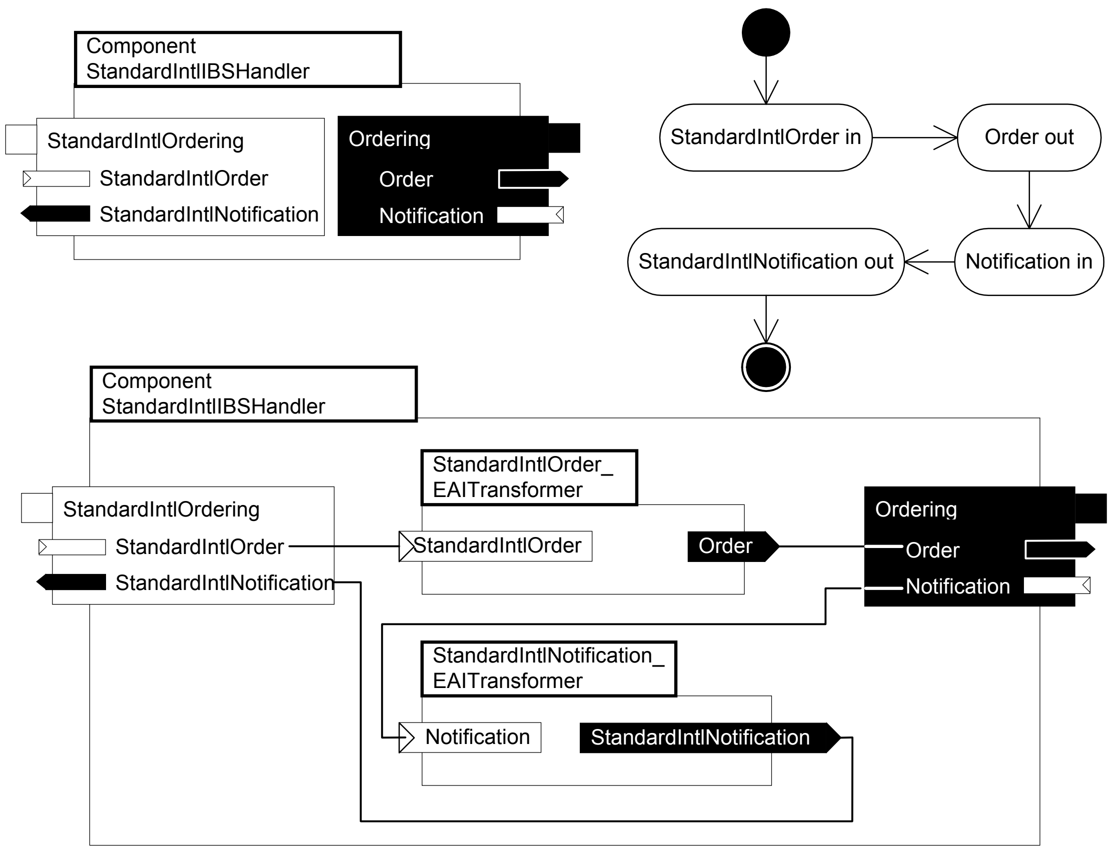



*Figure 11-14* Detail of StandardInternationalIMIBSHandler Component

# Part 5- Implementation Mappings

## Contents

This section contains the following chapters.



The profile presented in this specification is intended to provide the basis for modeling EAI architectures, largely at a logical level. However, the implementation of such an architecture requires, of course, the use of various technologies and tools appropriate to integration, such as message brokers. This section presents a selection of mappings of the modeling approaches of the profile into such implementation technologies. The set of technologies discussed here is by no means an exhaustive set of those applicable to EAI, but is simply intended to demonstration how the profile is usable with such technologies.

# Mapping to WebSphere MQ Integrator 12

WebSphere MQ Integrator (WMQI — formerly known as MQSeries Integrator) is IBM's *message broker* product, addressing the needs of business and application integration through management of information flow. It provides services that allow you to:

- Route a message to several destinations, using rules that act on the contents of one or more of the fields in the message or message header.
- Transform a message, so that applications using different formats can exchange messages in their own formats.
- Store and retrieve a message, or part of a message, in a database.
- Modify the contents of a message (for example, by adding data extracted from a database).
- Publish a message to make it available to other applications. Other applications can specify subscriptions that govern receipt of publications related to topics or topic ranges, optionally qualified by SQL-style filters based on message content.

These services exploit the message-oriented middleware (MOM) capability provided by the MQSeries and WebSphere MQ products.

This chapter presents a mapping from the EAI modeling elements to implementation elements; this is intended to show how an architectural model can be mapped to a more detailed implementation level.

## 12.1 WebSphere MQ Messaging

WebSphere MQ is IBM's new name for MQSeries.

### 12.1.1 WebSphere MQ Messages

WebSphere MQ messages are modeled as classes that conform to the ContentFormat stereotype. The most abstract version of this models the message as consisting of a header, which is content class MQMD (MQSeries Message Descriptor), and a body which is unconstrained. The MQMD contains the fundamental information required to allow efficient manipulation of a message by the WebSphere MQ messaging system, such as message expiry information and message identifier. The application-data portion of the message is effectively unconstrained, although a message type indicator within the MQMD can be used to indicate what format the message application data conforms to so that it can be checked at runtime.

Where more information is required for the middleware that is responsible for processing a message, extended header information has been defined. A few examples of these extended message formats are shown in Table 12-1; they include the message format expected by the WebSphereMQ CICS and IMS bridges, which enable intercommunication with applications running in CICS and IMS respectively, and the message format used by WMQI for Publish/Subscribe intercommunication.

One point to note about WebSphere MQ messages, which is correctly modeled by the structure shown, is that all of the more complex message types can, if desired, be treated as though they were simple WebSphere MQ messages. In this case, the extended header information is treated as part of the application data of the message.

The MQRFH2 message header is extensible, in that it allows arbitrary name/value data to be held in the header. In addition to mandatory fields contained within the header, it may also contain any number of 'NameValue' sections, which in turn may contain 'Folders.' Each folder may only contain data of the form *name=value*. Since messages are flattened structures, each of the associations between header, folder, and namevalue data is ordered, in that a sequence of values and structures can be reproducibly built from a message, though this ordering is not normally relied on to convey additional information.

*Table 12-1* WebSphere MQmessage classes

| **Class name** | **Parent class** | **Stereotype** | **Description** |
|---|---|---|---|
| WMQ Message | NA | ContentFormat | The WMQMessage is a specialization of the ContentFormat stereotype. It is the base format used by all WebSphere MQ applications. The message body is unconstrained.<br>The message header, known as MQMD is fully documented in the WebSphere MQ "Programming Reference Manual." |
| WMQCICS Bridge Message | WMQMessage | ContentFormat | Used in communication with the WebSphere MQ CICS Bridge. |
| WMQIMS Bridge Message | WMQMessage | ContentFormat | Used in communication with the WebSphere MQ IMS Bridge. |

Table 12-1 WebSphere MQmessage classes

| | | | |
|---|---|---|---|
| WMQI Message | WMQMessage | ContentFormat | Many WMQI message processing nodes can take advantage of information contained in an extended header, known as the MQRFH2.<br>Full details of the MQRFH2 header are given in the WebSphere MQ Integrator “Programming Reference Manual.” |
| WMQI Control Message | WMQIMessage | ContentFormat | The WMQIControlMessage class is a subclass of WMQIMessage. It allows control messages (such as add, cancel, and change a subscription).<br>Full details of command messages are given in the WebSphere MQ Integrator “Programming Reference Manual.” |

## 12.1.2 WebSphere MQ Message Queuing

WebSphere MQ queues are modeled as classes with the Queue stereotype. They can only hold messages that are in the WMQMessage format. The attributes of each class are not listed here, but are specified in the WebSphere MQ “Application Programming Guide.”

Table 12-2 WebSphere MQ Queue Stereotypes

| Class | Parent class | Stereotype | Constraint | Description |
|---|---|---|---|---|
| WMQQueue | NA | Queue | | WebSphere MQ message queue. Parent for all WebSphere MQ queue classes. |
| WMQLocal Queue | WMQQueue | Queue | Holds messages of class WMQMessage (or subclasses). | A physical queue owned by a particular queue manager. |
| WMQRemote Queue | WMQQueue | Queue | Must refer to a queue that is owned by a different queue manager. | A remote queue definition. Specifies the name and location of a queue owned by another queue manager. |
| WMQAlias Queue | WMQQueue | Queue | Must refer to a queue that is owned by the same queue manager. | An alias for another queue (a local queue) owned by the same queue manager. |

WebSphere MQ provides for two different indirection mechanisms, the queue *Alias*, which simply allows a queue to be referred to by a different name, and a *Remote Queue* definition, which identifies a queue managed by a different queue manager. The class diagram for alias queue and remote queue is given in Figure 12-1.

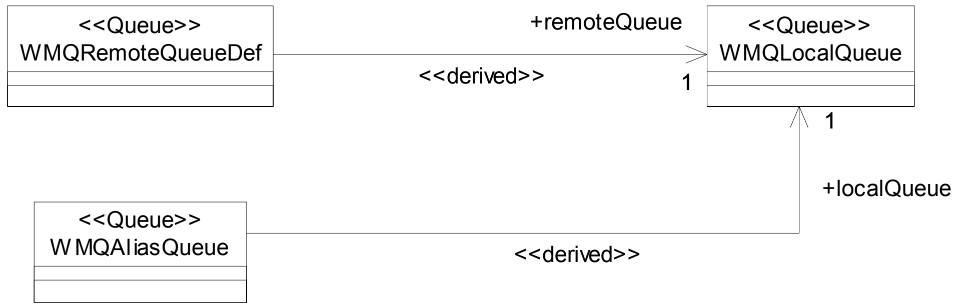



*Figure 12-1* WMQRemoteQueue and WMQAliasQueue

At runtime, the WebSphere MQ messaging infrastructure always resolves alias and remote queue definitions to a single local queue by following their 'remoteQueue' or 'localQueue' associations. Consequently, when specifying an EAI design that uses WebSphere MQ queues, the queue names used by the sender and receiver of a message need not match, but they must resolve to the same local queue.

## 12.2 WebSphere MQ Integrator Message Flows

### 12.2.1 Summary

Message routing and transformation is achieved within WMQI by constructing a *message flow*. This is done using a graphical tool, which allows operators to be joined together as *nodes* in a directed graph. A set of subclasses of WMQIPrimitiveNode is provided to perform tasks such as a message format conversion, a computation or a database operation; these are modeled as classes with the PrimitiveOperator stereotype. Message flows are modeled in the profile as classes with the CompoundOperator stereotype.

Top-level message flows are initiated via the receipt of a message on a message queue. They may invoke primitive nodes and nested message flows, which appear as *CompoundNodes* in the tool.

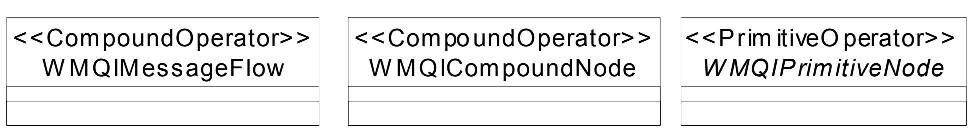



*Figure 12-2* Summary of the main usage of operator stereotypes

## 12.2.2 WMQIMessageFlow

### Description

WMQIMessageFlow models the outermost level of composition. At this outermost level, processing is initiated by the receipt of a message on a queue, as represented by WMQIInputNode. Consequently, an instance of WMQIMessageFlow must have at least one WMQIInputNode. This (see Figure 12-3) has the QueuedSource stereotype. Output may be produced by one of three different node classes: WMQIOutputNode, WMQIPublish, or WMQIReply. All of these nodes communicate externally using message queues. Consequently, the terminals (the view from the outside) of a message flow are required to have the QueuedTerminal stereotype.

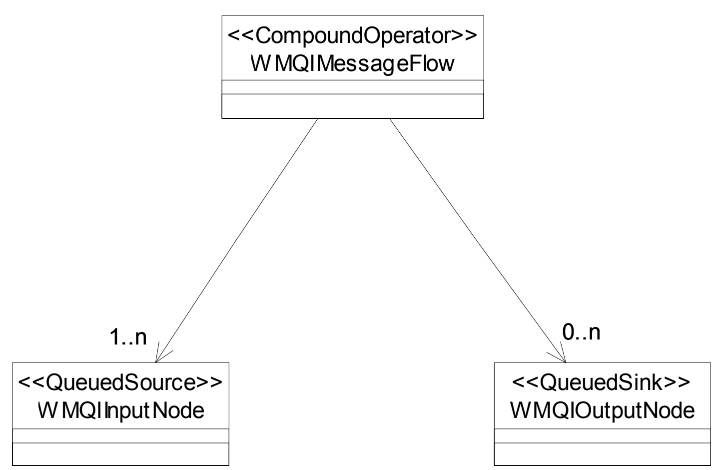



*Figure 12-3* WMQIMessageFlow

### Constraints

All links between the nodes that are contained in the message flow are synchronous.

WMQIMessageFlow must have at least one WMQIInputNode.

The external terminals of a WMQIMessageFlow have stereotype QueuedTerminal.

The external terminal that represents publication has, in addition, the stereotype PublicationTerminal.

WMQIMessageFlow can contain only WMQICompoundNode, WMQIPrimitiveNode, or its subtypes.

WMQIMessageFlow may *not* contain other WMQIMessageFlows (though a WMQIMessageFlow may invoke another WMQIMessageFlow by sending a message to the appropriate queue).

### 12.2.3 WMQICompoundNode

WMQICompoundNode models all levels of composition *inside* WMQIMessageFlow, exploiting the composition mechanism inherited from the FCM in the EAI Integration metamodel. Processing is initiated by sending a message to one of its terminals. Inside the compound node, this results in the emission of a message by a WMQIInputTerminalNode. Consequently, a WMQICompoundNode must have at least one WMQIInputTerminalNode. The results of message processing are propagated via WMQIOutputTerminals.

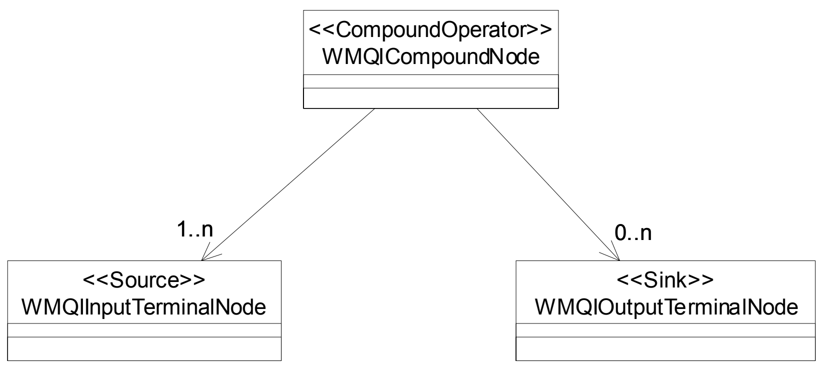



*Figure 12-4* Compound and primitive nodes in WMQI

#### *Constraints*

A WMQICompoundNode can contain WMQIPrimitiveNodes (and subclasses) and WMQICompoundNodes.

WMQICompoundNode may *not* contain a WMQIMessageFlow.

WMQICompoundNode does not have queued terminals.

All links between the nodes contained in a WMQICompoundNode have synchronization=synchronous.

### 12.2.4 WMQIPrimitiveNode

#### *Description*

WMQIPrimitiveNode is the (abstract) parent class for all WebSphere MQ Integrator message processing nodes.

#### *Constraints*

Primitive nodes all expect to receive and process messages that are of the WMQMessage class.

### 12.2.5 Supplied WMQIPrimitiveNodes

The WMQIPrimitiveNodes are modeled as classes and are listed in the table below with the appropriate stereotype from the UML Profile for EAI.

The table does not specify the attributes of these classes; the properties of these nodes are specified in the IBM WebSphere MQ “Using the Control Center” manual (IBM document number SC34-5602). Each of these properties may be represented as an attribute of the appropriate type for each class.

The interface required to allow further message processing nodes to be constructed is published by IBM.[1]

*Table 12-3* Mapping of WMQI primitive nodes to classes with stereotypes from the UML profile for EAI

| Class name | Parent Class | Stereotype | Constraint | Description |
|---|---|---|---|---|
| WMQI Publication | WMQI PSService | Publication Operator | Output terminal is a QueuedPublication Terminal. Input terminal is expect message type WMQIMessage. | The Publication node filters and transmits the output from a message flow to subscribers who have registered an interest in a particular set of topics. The Publication node must always be an output node of a message flow and have no output terminals of its own. |
| WMQI PSService | WMQI PrimitiveNode | Primitive Operator | NA | The PS Service node allows for the interception of publications after they have passed the subscription filters. |
| WMQICheck | WMQI PrimitiveNode | Filter | NA | A Check node compares the format of a message arriving on its input terminal with its message-type specification. |
| WMQI Compute | WMQI PrimitiveNode | Transformer | NA | The Compute node constructs an output message. The elements of the output message can be defined using an SQL expression, and can be based on elements of both the input message and data from an external database. |
| WMQI Database | WMQI PrimitiveNode | Primitive Operator | NA | The Database node applies an SQL expression to an external database table. Data from the message input to this node can be used in the SQL expression. |
| WMQI DataDelete | WMQI DatabaseNode | Primitive Operator | NA | A DataDelete node deletes one or more rows from a table in a specified database. Data from the input message can be used as part of the expression that determines which rows are deleted. |
| WMQI DataInsert | WMQI Database | Primitive Operator | NA | A DataInsert node inserts a new row into a database table. Data from the input message can be included in the database insert expression. |
| WQMIDataU pdate | WMQI Database | Primitive Operator | NA | A DataUpdate node updates one or more rows of data in a specified database. Data from the input message can be used as part of the expression that determines which rows are updated. |
| WQMIWareh ouse | WMQI Database | Primitive Operator | NA | A Warehouse node saves a copy of the input message in a database table by inserting it in a new row. |

1. WebSphere MQ Programming Guide *SC34-5603*

*Table 12-3* Mapping of WMQI primitive nodes to classes with stereotypes from the UML profile for EAI

| | | | | |
|---|---|---|---|---|
| WQMI Extract | WMQI Compute | Transformer | NA | The Extract node derives an output message from an input message. The output message comprises only those elements of the input message that are specified for inclusion when configuring the Extract node. |
| WQMIFilter | WMQI PrimitiveNode | Filter | NA | A Filter node routes a message according to message content using a filter expression specified in SQL. The filter expression can include elements of the input message or message properties. It can also use data held in an external database. The output terminal to which the message is routed depends on whether the expression is evaluated to true, false, or unknown. |
| WMQIInput | WMQI PrimitiveNode | QueuedSource | NA | Receives a WebSphere MQ message from a specified queue. |
| WQMI Output | WMQI PrimitiveNode | QueuedSink | NA | Sends a WebSphere MQ message to the specified target queues. |
| WMQIReply | WMQIOutput | QueuedSource | NA | Sends a reply message to the WebSphere MQ queue specified in the message header. |
| WQMIFlow Order | WMQI PrimitiveNode | Primitive Operator | NA | The FlowOrder node enables you to specify the order in which each message is propagated to each (of two) output terminals. The message is only propagated to the second output terminal if propagation to the first output terminal is successful. |
| WQMIReset Content Descriptor | WMQI PrimitiveNode | Transformer | NA | The ResetContentDescriptor node takes the bit stream of the input message and reparses it using a different message template from the same or a different message dictionary. The node can reset any combination of message domain, set, type, and format. |
| WMQITry Catch | WMQI PrimitiveNode | Primitive Operator | NA | The TryCatch node provides a special handler for exception processing. The input message is initially routed on the try terminal of this node. If an exception is subsequently thrown by a downstream node, it is caught by this node, which then routes the original message to its catch terminal. |
| WMQI Throw | WMQI PrimitiveNode | Primitive Operator | NA | The Throw node provides a mechanism for throwing an exception within a message flow. The exception might be caught and processed by a preceding TryCatch node within the message flow, or handled by the MQInput node. |
| WMQI Aggregate Reply | WMQI PrimitiveNode | Aggregator | NA | The AggregateReply node holds related messages until either a complete set has arrived (according to a specified condition) or a time limit has elapsed. |

## *12.2.6 The Role of the WMQI message-broker topology*

A set of WMQI message brokers is interconnected and governed by the WMQI Configuration Manager, which we represent by the class WMQIntegrator. WMQIntegrator owns all executing WMQIMessageFlows, as shown in Figure 12-5. The Configuration Manager deploys these to selected message brokers. The set of WMQI message brokers also acts as a SubscriptionOperator, allowing subscriptions to be added to, and removed from, the subscription table (see Table 12-4). The topology is governed by the Configuration Manager. All WMQIPublication nodes that are owned by message flows in the same broker topology share the same subscription table. (The implementation optimizes the distribution of the subscription table.)

*Table 12-4* WMQIntegrator class definition table

| **Class** | **Stereotype** | **Constraint** | **Description** |
|---|---|---|---|
| WMQIntegrator | Subscription Operator | Input terminal is a QueuedInputTerminal. Expects to receive messages in WMQICommandMessage format. | The WMQI message broker topology when acting as a subscription operator.<br><br>Subscriptions are added, removed, and updated on WMQIntegrator by sending a message that conforms to the WMQICommandMessage format to the WMQI command queue. |

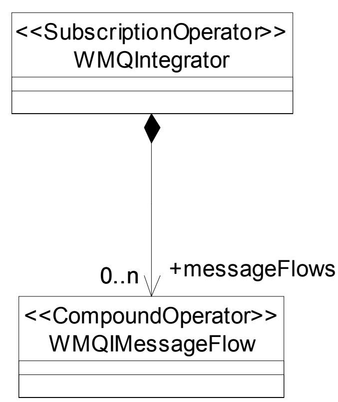



*Figure 12-5* WMQIntegrator class diagram

# Java Message Service (JMS) 13

The Java Message Service (JMS)[1] is part of the 1.3 release of the J2EE™ platform specification.[2] It specifies a point-to-point (PTP) domain and a publish-subscribe (Pub/Sub) domain. The JMS entities of interest in modeling are destinations, message producers and message consumers. These are summarized in the table below.

| JMS Parent | PTP Domain | Pub/Sub Domain |
|---|---|---|
| Destination | Queue | Topic |
| MessageProducer | QueueSender | TopicPublisher |
| MessageConsumer | QueueReceiver, QueueBrowser | TopicSubscriber |

These entities are all defined in the EAI Integration metamodel, except that the distinction between receivers and browsers is not made. A JMS QueueReceiver receives a message destructively from a queue, whereas a JMS QueueBrowser leaves it on the queue so that it may be read again.

## 13.1 PTP Domain

A JMS client acting as a sender creates one or more JMS QueueSender objects and sends messages on them. These are modeled as a class JMSQueueSender with stereotype QSource.

1.For the JMS 1.2 specification see http://java.sun.com/products/jms/

2.At the time of writing, J2EE 1.3 is still in draft. See http://java.sun.com/j2ee/

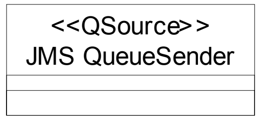



*Figure 13-1* JMS QueueSender

A JMS client acting as a receiver creates one or more JMS QueueReceiver or QueueBrowser objects and listens on them. A JMS QueueReceiver or QueueBrowser object may include a JMS message selector, which has the effect of a local EAI filter.

In order to model this optional filtering behavior, QueueReceiver and QueueBrowser are both modeled as <<CompoundOperator>> classes, each with a single queued input terminal. The composition that defines them contains a class QDataIn of stereotype <<QSource>>. The class QDataIn makes messages received at the input terminal available to the JMS Message Selector (if there is one) but does not remove them from the queue. The *emit* operation of the JMSMessageSelector (a <<Stream>>) emits the message from the stream, provided it passes the chosen filter condition, and passes it on to the sink. The <<QSource>> QDataIn and the stream both share the same queue resource. This means that messages remain on the input queue unless they are explicitly sent to the sink.

The difference between QueueReceiver and QueueBrowser lies in the behavior of the stream. For QueueBrowser, the stream does not remove messages; it proceeds forward through them, but they remain available for other receivers and browsers. For QueueReceiver, the stream removes those messages that pass the filter condition of the JMSMessageSelector; the remaining messages are available for access by other receivers and browsers.

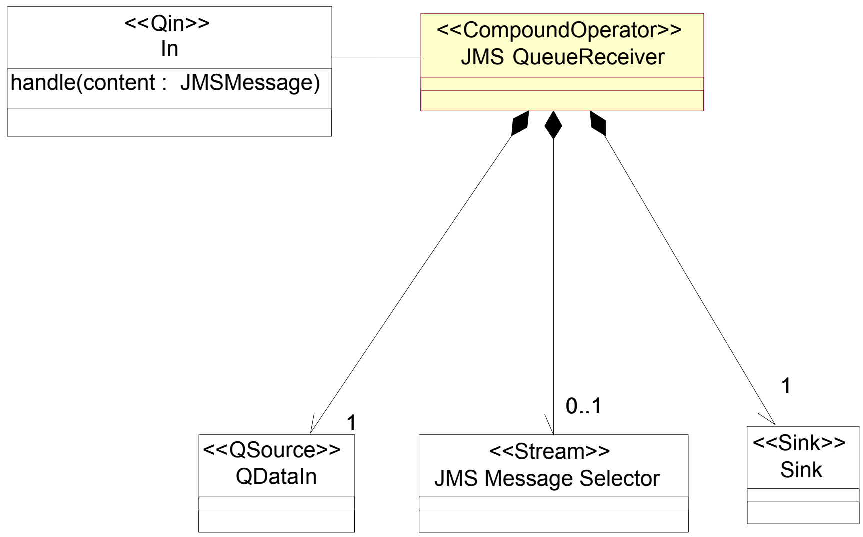



*Figure 13-2* JMS QueueReceiver

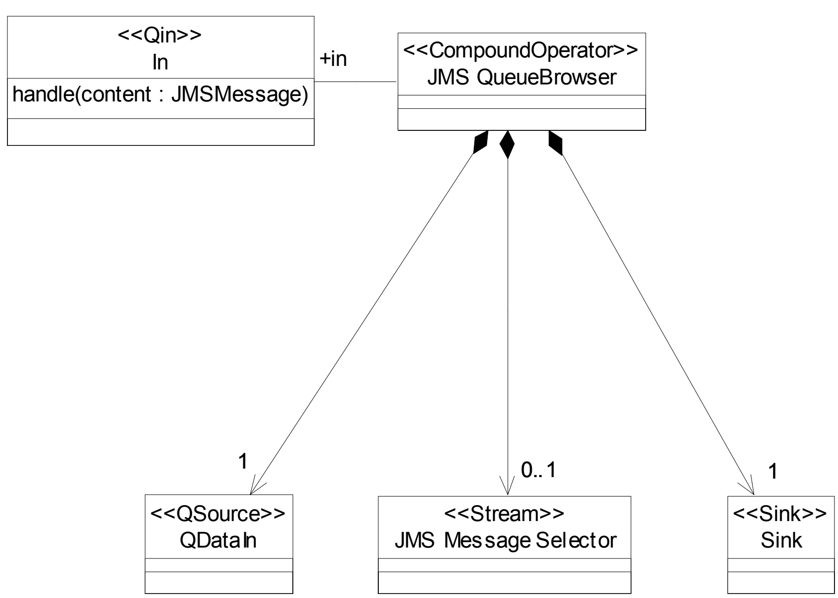



*Figure 13-3* JMS QueueBrowser

## 13.2 Pub/Sub Domain

A JMS client acting as a subscriber registers its interest in topics by creating one or more JMS TopicSubscriber objects and listening on them. To model this in the EAI profile, we separate the creation of a JMS TopicSubscriber from the activity of listening to the topic.

We model the 'listener' aspect as a class JMSSubscriberListener of stereotype Sink that expects a JMSMessage as its input.

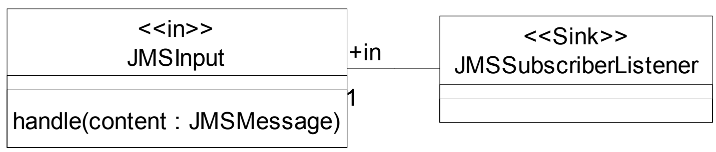



*Figure 13-4* A JMSSubscriberListener expects incoming messages

A JMS TopicSubscriber object refers to a JMS Topic object, and it may include a JMS message selector. A JMS Topic may refer to several EAI topics.

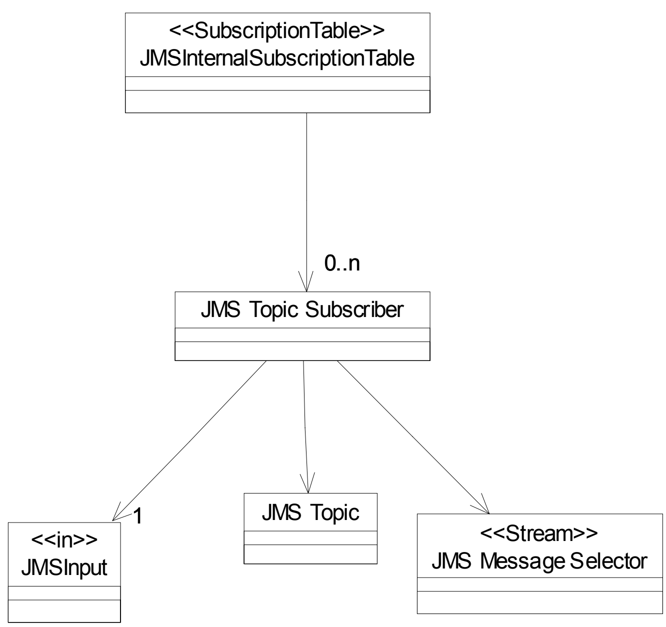



*Figure 13-5* Model for the content of the JMS subscription table

Creating a JMS subscriber object causes a subscription to be registered with the JMS infrastructure. We model the element that registers the subscription as a JMSTopicSubscriberCreator of stereotype <<source>> that sends a subscription to the JMS subscription infrastructure.

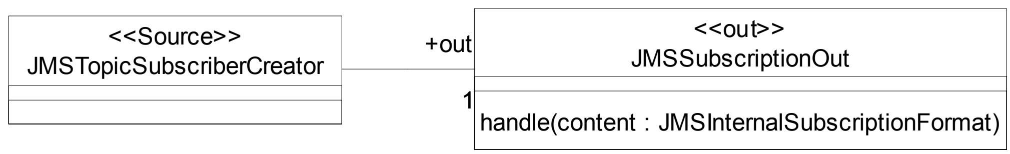



*Figure 13-6* JMSTopicSubscriberCreator

We model the subscription infrastructure via a class JMSSubscriptionInfrastructure of stereotype <<SubscriptionOperator>>. This expects a message of the arbitrary 'JMSInternalSubscriptionData' format.

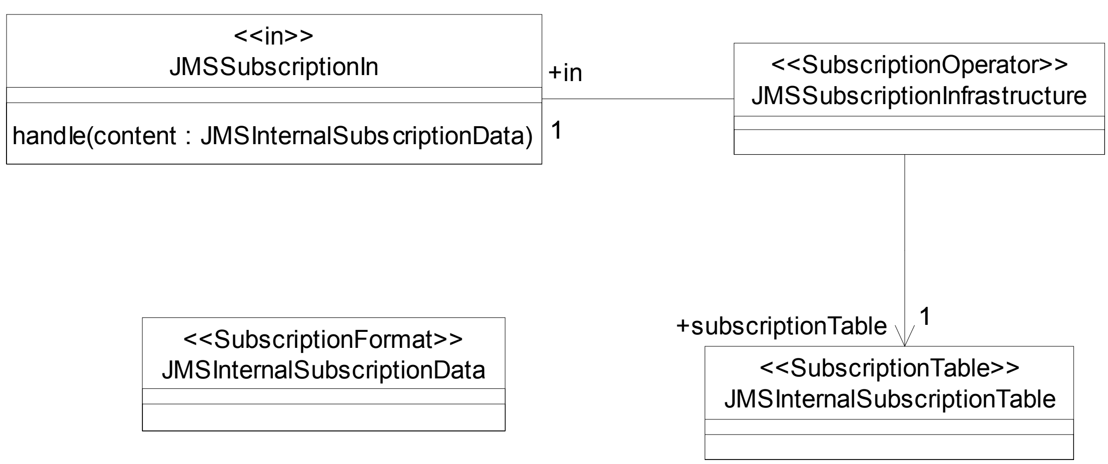



*Figure 13-7* JMSSubscriptionInfrastructure

A JMS client acting as a publisher creates one or more JMS TopicPublisher objects that identify topics via JMS Topic objects. The publisher produces messages and sends them on one or more topics, using the associated JMS TopicPublisher object.

This has the effect of sending them to a PublicationOperator (Figure 13-9), which forwards them to the appropriate EAI destinations; these can include JMS subscribers.

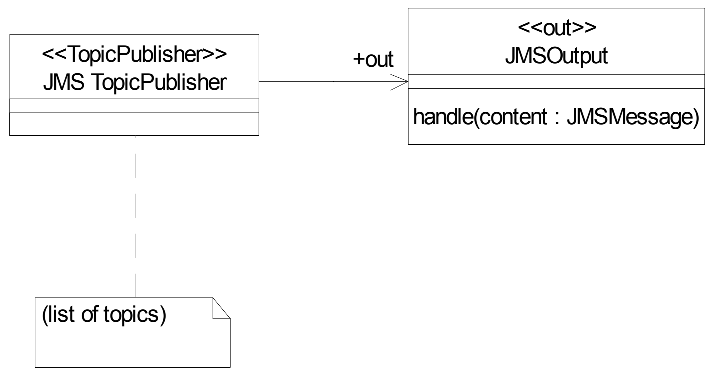



*Figure 13-8* A JMS TopicPublisher

We model the existence of a publication mechanism via the class JMSPublicationInfrastructure of stereotype <<PublicationOperator>>. This is not a separable element of JMS, but is part of the JMS infrastructure. All JMSTopicPublishers for a given JMS environment should be connected to the same JMSPublicationInfrastructure.

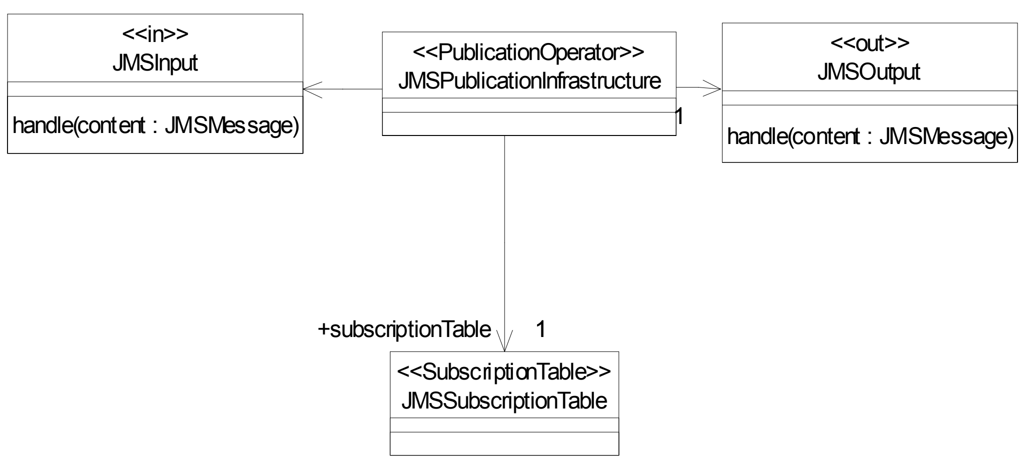



*Figure 13-9* JMSPublicationInfrastructure

# Language Metamodels 14

## 14.1 COBOL Metamodel

The COBOL metamodel is used by enterprise application programs to define data structures (semantics), which represent connector interfaces.

The goal of this COBOL model is to capture the information that would be found in the Data Division. This model is intended to be used only as read-only to convert COBOL data division into its XML equivalent. This model is not intended to be used as a converter from XML code into a COBOL data division equivalent. The following figures illustrate the classes that constitute the COBOL metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

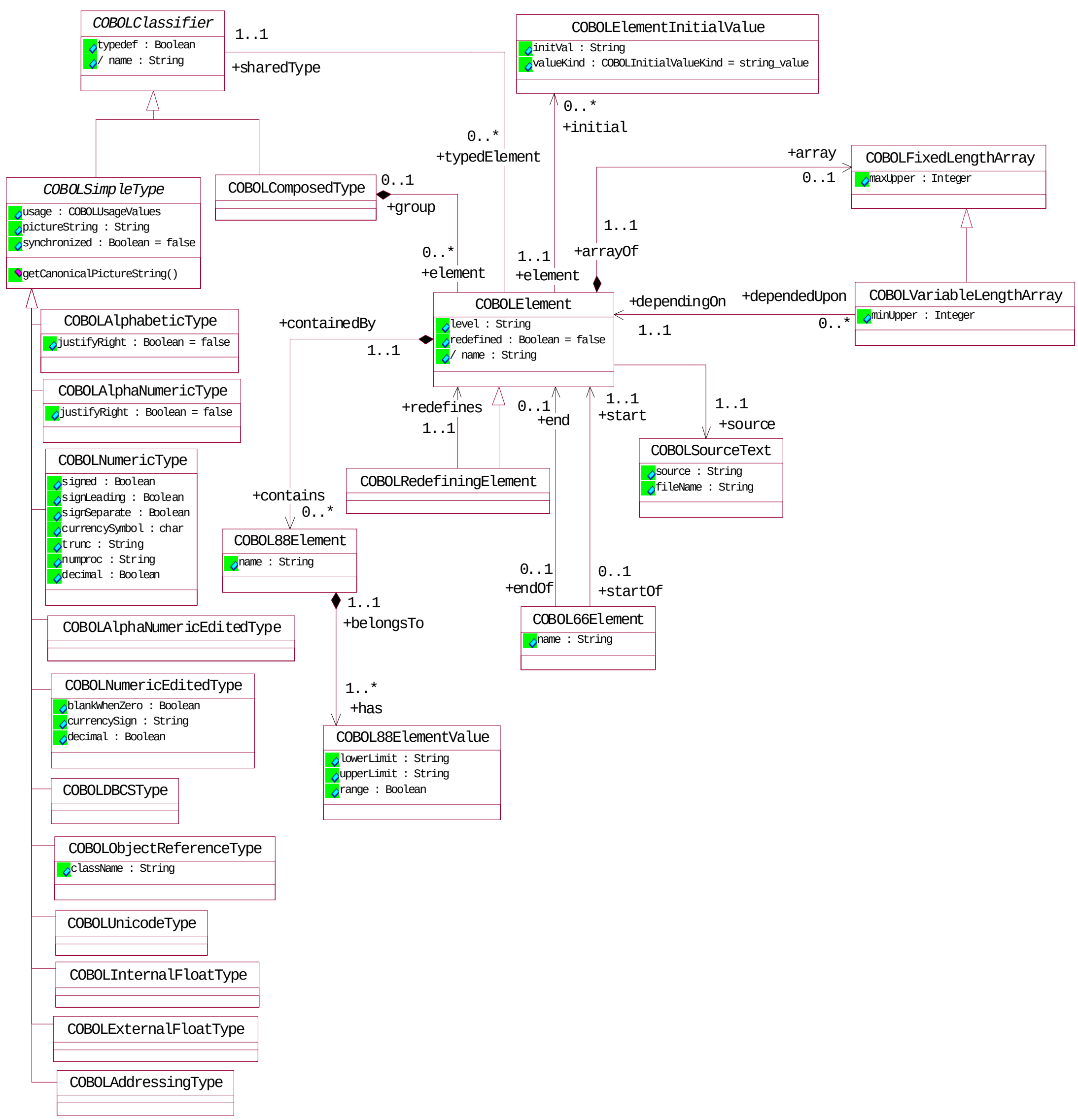



*Figure 14-1* COBOL Metamodel

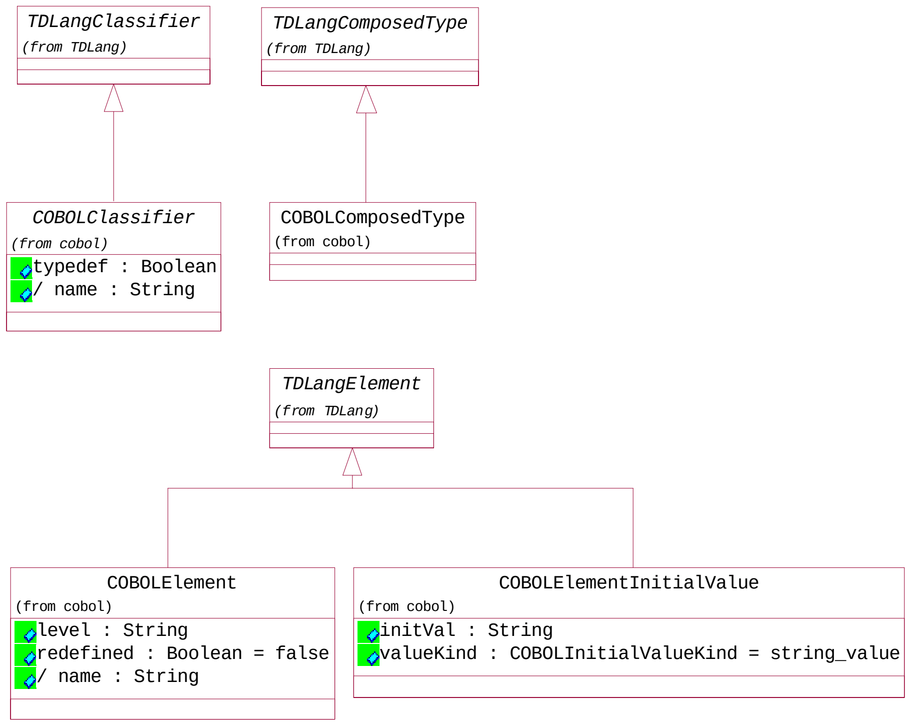



*Figure 14-2* TDLang to COBOL

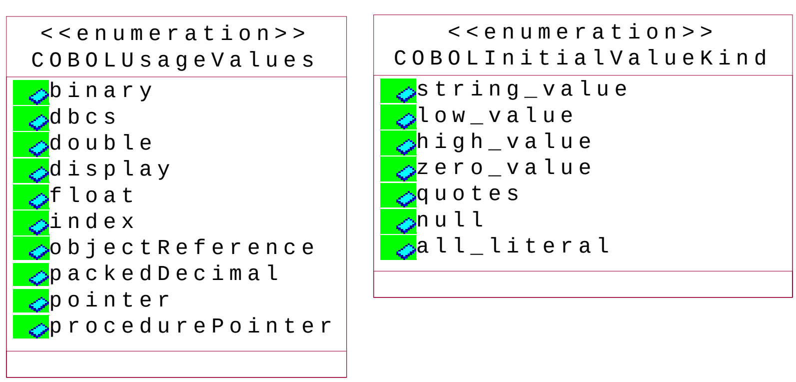



*Figure 14-3* COBOL Stereotypes

## 14.1.1 COBOL Metamodel Descriptions

### 14.1.1.1 COBOL66Element

COBOL66Element represents the COBOL 66 data level.

For example:

```
01 DATA-GROUP  PIC 9.
     03 DATA1 VALUE 1.
     03 DATA2 VALUE 2.
     03 DATA3 VALUE 3.
66 SUB-DATA RENAMES DATA1 THROUGH DATA2.
66 AKA-DATA3 RENAMES DATA3.
```

In this example SUB-DATA refers to contents in DATA1 and DATA2.

### 14.1.1.2 COBOL88Element

COBOL88Element represents the COBOL 88 data level.

For example:

```
1 TESTX PIC .
   88 TRUEX VALUE 'T' 't'.        *(TRUEX has 2 values)
   88 FALSEX VALUE 'F' 'f'.       *(FALSEX has 2 values)
```

Where TRUEX and FALSEX are condition names for the TESTX variable if value equals ('T' or 't') or ('F' or 'f'), respectively. So if TESTX = 'T' or 't' then TRUEX = TRUE and FALSEX = FALSE; If TESTX = 'F' or 'f' then FALSEX = TRUE and TRUEX = FALSE.

### 14.1.1.3 COBOL88ElementValue

COBOL88ElementValue represents the values specified by COBOL88Element.

### 14.1.1.4 COBOLAddressingType

COBOLAddressingType is used for index values, pointer values, and procedure pointer values.

### 14.1.1.5 COBOLAlphabeticType

COBOLAlphabeticType represents a picture string consisting of alphabetic characters.

### 14.1.1.6 COBOLAlphaNumericEditedType

COBOLAlphaNumericEditedType represents a picture string consisting of either alphabetic or alphanumeric type and at least one blank (B), zero (0), or slash (/).

### 14.1.1.7 COBOLAlphaNumericType

COBOLAlphaNumericType represents a picture string consisting of alphabetic and numeric characters.

### 14.1.1.8 COBOLClassifier

COBOLClassifier represents all data types of the COBOL metamodel. COBOLClassifier is the parent class of COBOLComposedType and COBOLSimpleType.

### 14.1.1.9 COBOLComposedType

COBOLComposedType represents a nested declaration that contains additional elements. COBOLComposedType has a single aggregation to include all the elements that are part of this composition.

### 14.1.1.10 COBOLDBCSType

COBOLDBCSType represents double byte character strings whose code is represented by 16 bits instead of 8 bits.

### 14.1.1.11 COBOLElement

COBOLElement represents data elements in the COBOL metamodel.

### 14.1.1.12 COBOLElementInitialValue

COBOLElementInitialValue stores the value assigned to a COBOLElement at the time storage is allocated for it.

### 14.1.1.13 COBOLExternalFloatType

COBOLExternalFloatType represents how COBOL floating points are displayed to the user.

### 14.1.1.14 COBOLFixedLengthArray

COBOLFixedLengthArray represents an array declared as OCCURS N TIMES.

### 14.1.1.15 COBOLInitalValueKind

COBOLInitalValueKind is an enumeration of types supported in an initialized element.

### 14.1.1.16 COBOLInternalFloatType

COBOLInternalFloatType represents COBOL's internal float data type.

### 14.1.1.17 COBOLNumericEditedType

COBOLNumericEditedType represents formatted numeric values. COBOLNumericEditedType values can be decorated with characters such as decimal point (.), dollar sign ($), and arithmetic signs (+,-,*,/).

### 14.1.1.18 COBOLNumericType

COBOLNumericType represents a numeric data number, including the implied decimal point and operational sign. COBOLNumericType can represent binary, packed decimal, and zoned decimal types.

### 14.1.1.19 COBOLObjectReferenceType

COBOLObjectReferenceType represents an object declared in COBOL as USAGE OBJECT REFERENCE.

### 14.1.1.20 COBOLRedefiningElement

COBOLRedefiningElement represents an element declared with the REDEFINES clause. COBOLRedefiningElement allows different data description entries to describe the same computer storage area.

### 14.1.1.21 COBOLSimpleType

COBOLSimpleType is an abstract class that contains attributes shared by all simple types in the COBOL metamodel.

### 14.1.1.22 COBOLSourceText

This class contains the entire source code (including comments) and its associated line number.

### 14.1.1.23 COBOLUnicodeType

COBOLUnicodeType represents COBOL data declared in Unicode format.

#### 14.1.1.24 COBOLUsageValues

COBOLUsageValues is an enumeration of values supported in the USAGE clause.

#### 14.1.1.25 COBOLVariableLengthArray

COBOLVariableLengthArray represents an array declared as OCCURS DEPENDING ON.

## 14.2 PL/I Metamodel

The PL/I language metamodel is used by enterprise application programs to define data structures (semantics), which represent connector interfaces.

This language model for PL/I attempts to describe PL/I declares that have the storage class of either PARAMETER, STATIC, or BASED. CONTROLLED, AUTOMATIC, and DEFINED are not supported.

In the PL/I languages, extents (that is string lengths, area sizes, and array bounds) may, in general, be declared as constants, as expressions to be evaluated at run-time, as asterisks, or as defined via the REFER option; however, none of these choices are valid for all storage classes.

Based variables whose extents are not constant and not defined via the REFER option are excluded from this model, as are parameters whose extents are specified via asterisks.

The INITIAL attribute (which is not valid for parameters in any case) will be ignored by the model. The following figures illustrate the classes that constitute the PL/I metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

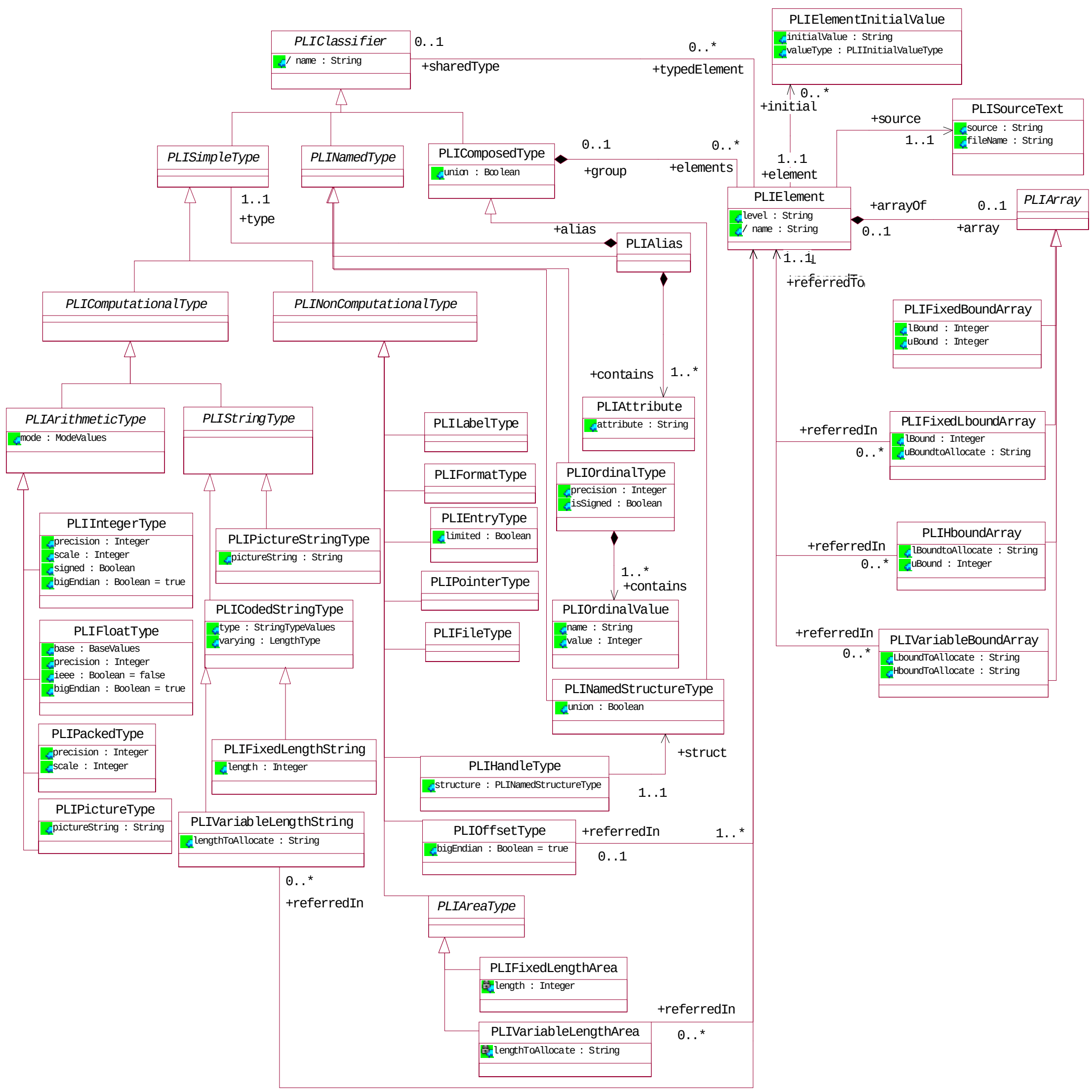



*Figure 14-4* PL/I Metamodel

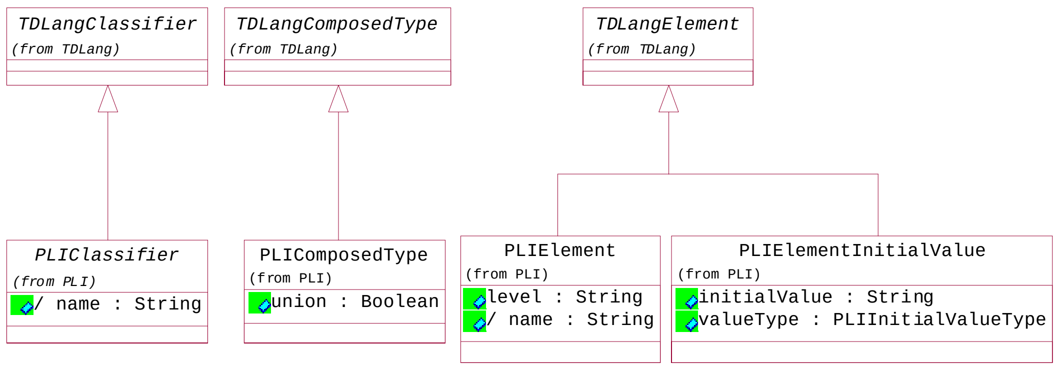



*Figure 14-5* TDLang to PL/I

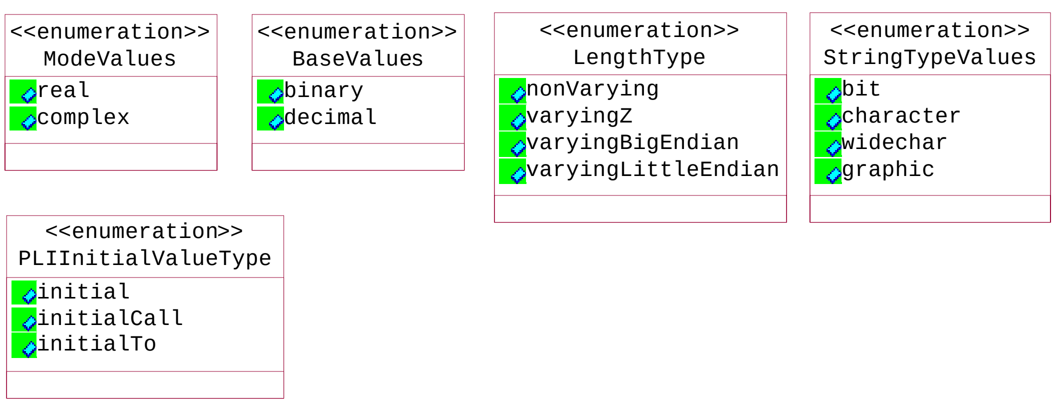



*Figure 14-6* PL/I Stereotypes

## 14.2.1 PL/I Metamodel Descriptions

### 14.2.1.1 PLIAlias

PLIAlias represents an alias defined for a collection of data attributes.

### 14.2.1.2 PLIAreaType

PLIAreaType represents an area variable that describes an area of storage reserved for the allocation of a based variable.

### 14.2.1.3 PLIArithmeticType

PLIArithmeticType represents data types that can be represented as rational numbers.

### 14.2.1.4 PLIArray

PLIArray represents an n-dimensional collection of elements that have identical attributes.

### 14.2.1.5 PLIBaseValues

Base Values is an enumeration of base values used by PLIFloatType.

### 14.2.1.6 PLIClassifier

PLIClassifier represents all data types of the PL/I metamodel.

### 14.2.1.7 PLICodedStringType

PLICodedStringType represents a character string data item that can contain any of the available set of characters.

### 14.2.1.8 PLIComposedType

PLIComposedType is a collection of member elements that can be structure, unions, or elementary variables and arrays. PLIComposedType has a single aggregation to include all the elements that are a part of this composition.

### 14.2.1.9 PLIComputationalType

PLIComputationalType represents types used in computations to produce a desired result. Arithmetic and string data types constitute computational data type.

### 14.2.1.10 PLIElement

PLIElement represents data elements in the PL/I metamodel.

### 14.2.1.11 PLIElementInitialValue

PLIElementInitialValue stores the value assigned to a PLIElement at the time storage is allocated for it.

### 14.2.1.12 PLIEntryType

PLIEntryType represents an entry constant or the value of an entry variable.

### 14.2.1.13 PLIFileType

PLIFileType represents the FILE attribute that specifies the associated file name or file variable.

### 14.2.1.14 PLIFixedBoundArray

PLIFixedBoundArray represents a fixed size array.

### 14.2.1.15 PLIFixedLboundArray

PLIFixedLboundArray represents an array whose lower bound is fixed.

### 14.2.1.16 PLIFixedLengthArea

PLIFixedLengthArea represents a PLIAreaType whose area size is fixed.

### 14.2.1.17 PLIFixedLengthString

PLIFixedLengthString represents a PLICodedStringType whose string length is fixed.

### 14.2.1.18 PLIFloatType

PLIFloatType represents numbers stored in floating-point format.

### 14.2.1.19 PLIFormatType

PLIFormatType represents a format list is to be used in a FORMAT statement.

### 14.2.1.20 PLIHandleType

PLIHandleType represents a variable as a pointer to a structure type.

### 14.2.1.21 PLIHboundArray

PLIHboundArray represents an array whose upper bound is fixed.

### 14.2.1.22 PLIInitialValueType

PLIInitialValueType is an enumeration of initial value types used by PLIElementInitialValue.

#### 14.2.1.23 PLIIntegerType

PLIIntegerType represents numbers stored in binary fixed-point format.

#### 14.2.1.24 PLILabelType

PLILabelType represents a label constant or the value of a label variable.

#### 14.2.1.25 PLILengthType

PLILengthType is an enumeration of length types supported by PLICodedStringType.

#### 14.2.1.26 PLIModeValues

PLIModeValues is an enumeration specifying the mode used by PLIArithmeticType.

#### 14.2.1.27 PLINamedStructureType

PLINamedStructureType represents a named structure. A structure is a collection of member elements that can be structure, unions, or elementary variables and arrays.

#### 14.2.1.28 PLINamedType

PLINamedType represents user-defined name types.

#### 14.2.1.29 PLINonComputationalType

PLINonComputationalType represents values used to control execution of a PL/I program.

#### 14.2.1.30 PLIOffsetType

PLIOffsetType represents an offset value relative to the locations of a base variable.

#### 14.2.1.31 PLIOrdinalType

PLIOrdinalType represents a named set of ordered values. The values of PLIOrdinalType are stored in PLIOrdinalValue.

#### 14.2.1.32 PLIOrdinalValue

PLIOrdinalValue stores the values specified by PLIOrdinalType.

### 14.2.1.33 PLIPackedType

PLIPackedType represents numbers stored in packed-decimal format.

### 14.2.1.34 PLIPictureStringType

PLIPictureStringType represents a fixed-length character data item, with the additional restriction that the data item can only contain characters from certain subsets of the complete set of available characters.

### 14.2.1.35 PLIPictureType

PLIPictureType represents numeric data held in character form.

### 14.2.1.36 PLIPointerType

PLIPointerType represents a pointer.

### 14.2.1.37 PLISimpleType

PLISimpleType is an abstract class that contains attributes shared by all simple types in the PL/I metamodel.

### 14.2.1.38 PLISourceText

This class contains the entire source code (including comments) and its associated line number.

### 14.2.1.39 PLIStringType

PLIStringType represents a sequence of contiguous characters, bit, widechars, or graphics that are treated as a single data item.

### 14.2.1.40 PLIStringTypeValues

PLIStringTypeValues is an enumeration of types supported by PLICodedStringType.

### 14.2.1.41 PLIVariableBoundArray

PLIVariableBoundArray represents an array whose upper and lower bound are both variable.

### 14.2.1.42 PLIVariableLengthArea

PLIVariableLengthArea represents a PLIAreaType whose area size is variable.

### 14.2.1.43 PLIVariableLengthString

PLIVariableLengthString represents a PLICodedStringType whose string length is variable.

## 14.3 C Metamodel

The C metamodel including C Main and User Types (i.e., user defined types) is a MOF Class instance at the M2 level.

The C metamodel is used by enterprise application programs to define data structures, that represent connector interfaces. The following figures illustrate the classes that constitute the C metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

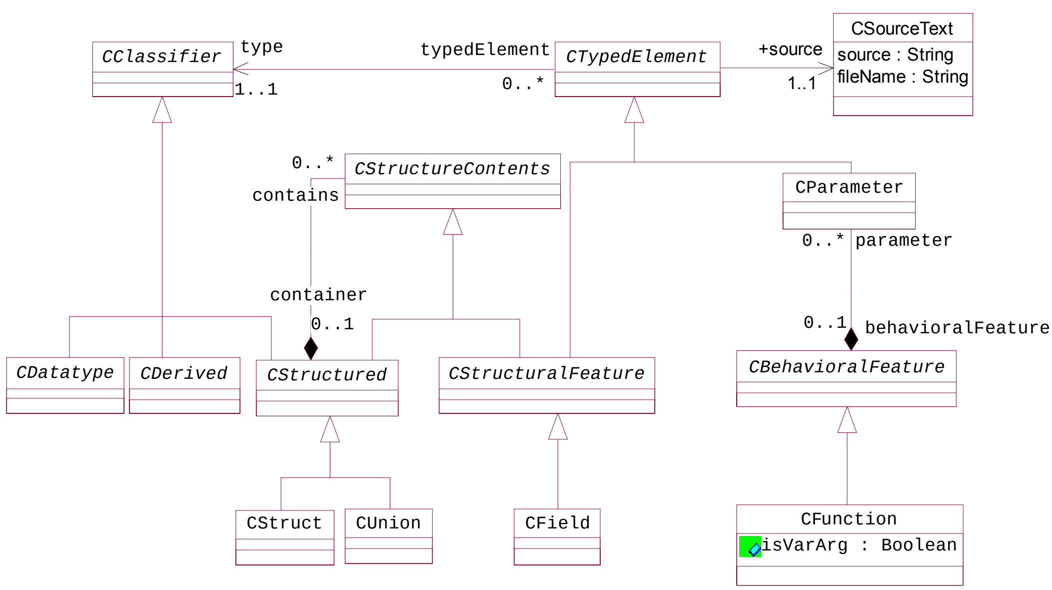



*Figure 14-7* C Metamodel

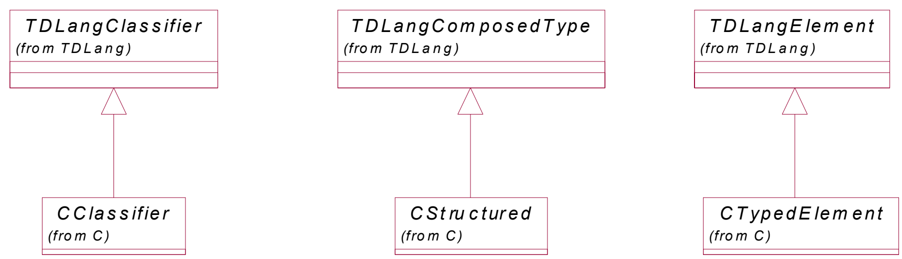



*Figure 14-8* TDLang to C

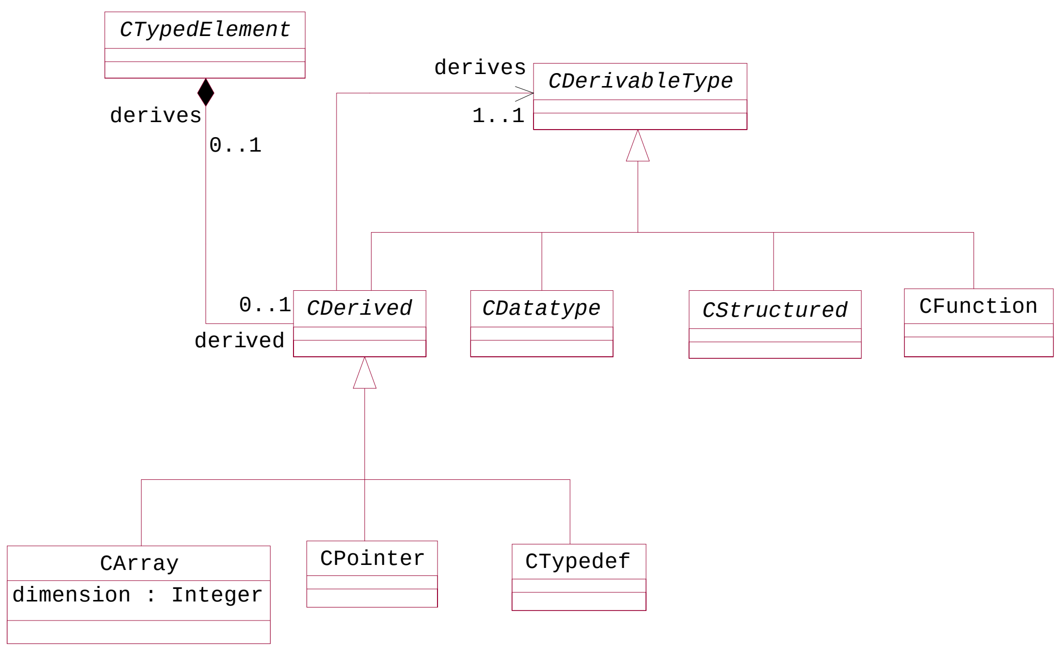



*Figure 14-9* C Derivation

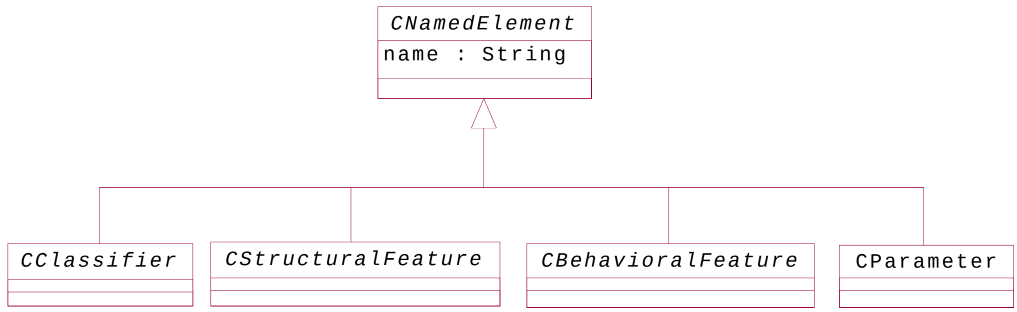



*Figure 14-10* C Names

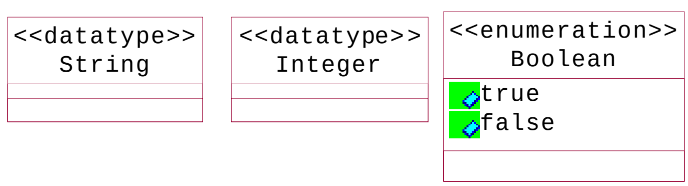



*Figure 14-11* C Datatype - Model Types

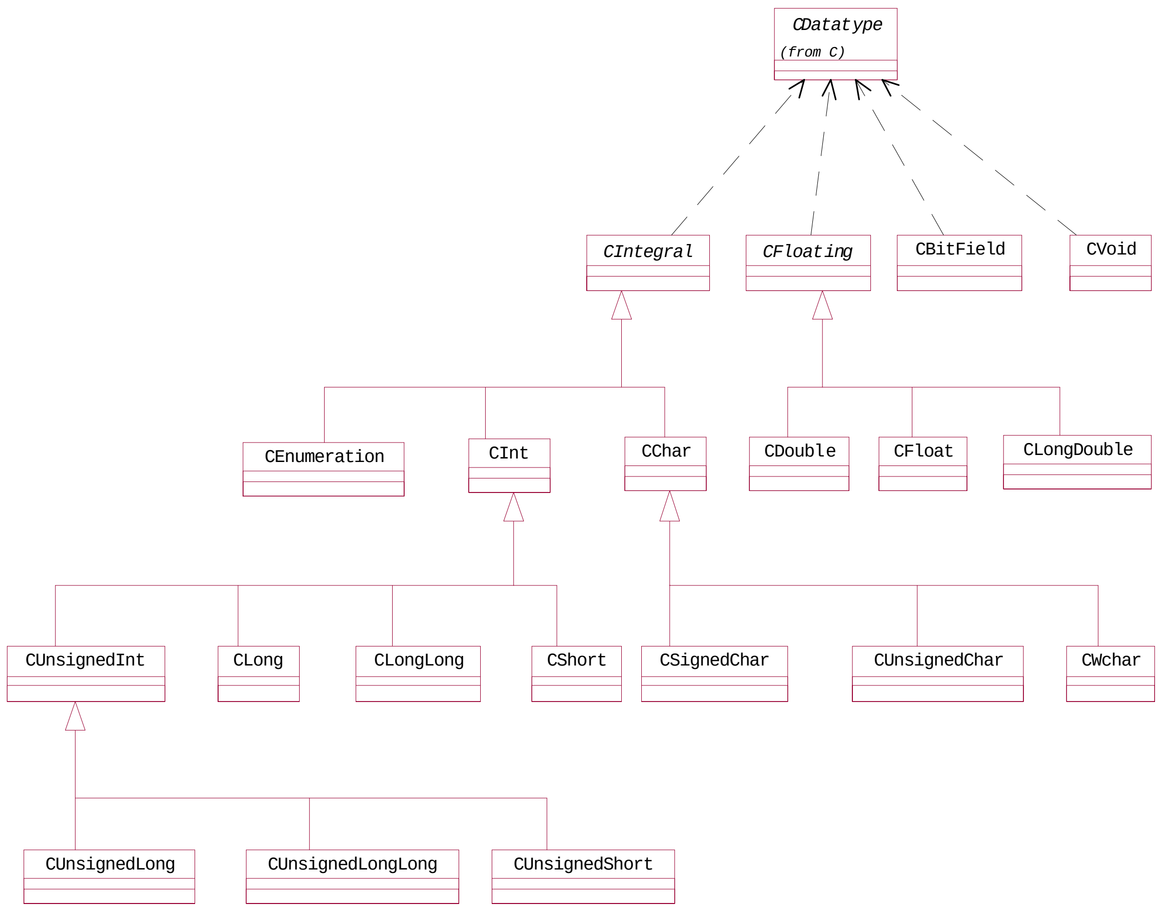



*Figure 14-12* C User Types

## 14.3.1 C Metamodel Descriptions

### 14.3.1.1 CArray

CArray represents an ordered group of data objects. CArray refers to each object as an element. All elements within an array have the same data type.

### 14.3.1.2 CBehavioralFeature

CBehavioralFeature represents dynamic characteristics of the ModelElement that contains it. CBehavioralFeature is both a Feature and a Namespace. CBehavioralFeature serves as the parent of CFunction.

#### 14.3.1.3 CClassifier

CClassifier represents all data types of the C metamodel. CClassifier is the parent class of C Derived types.

#### 14.3.1.4 CDatatype

CDatatype represents data types and native types.

#### 14.3.1.5 CDerivableType

CDerivableType represents datatypes that can be derived from CDatatype.

#### 14.3.1.6 CDerived

CDerived represents datatypes derived from CDatatypes.

#### 14.3.1.7 CField

CField represents attributes defined in an instance of the C metamodel.

#### 14.3.1.8 CFunction

CFunction represents functions defined in an instance of the C metamodel.

#### 14.3.1.9 CParameter

CParameter provides a means of communication with operations and CBehavioralFeature. A CParameter passes or communicates values of its defined type.

#### 14.3.1.10 CPointer

CPointer represents a derived datatype declared as a pointer.

#### 14.3.1.11 CSourceText

This class contains the entire source code (including comments) and its associated line number.

#### 14.3.1.12 CStruct

CStruct represents a structure declared as type struct.

#### 14.3.1.13 CStructuralFeature

CStructuralFeature represents static characteristics of the ModelElement that contains it. CStructuralFeature serves as the parent of CField.

#### 14.3.1.14 CStructureContents

CStructureContents represent structured data types and structural features.

#### 14.3.1.15 CStructured

CStructured is an abstract class that represents all structured data types of the C metamodel.

#### 14.3.1.16 CTypedef

CTypedef represents a derived datatype declared as type typedef.

#### 14.3.1.17 CTypedElement

CTypedElement represents data elements in the C metamodel.

#### 14.3.1.18 CUnion

CUnion represents a structure declared as type union.

## 14.4 C++ Metamodel

The C++ metamodel, based on the ANNOTATED C++ REFERENCE MANUAL book (authors: Margaret A. Ellis, Bjarne Stoustrup), 1990, is a MOF Class instance at the M2 level. The C++ metamodel consists of C++ Main, and Model Types. This metamodel inherits from the C Main metamodel. The following figures illustrate the classes that constitute the C++ metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

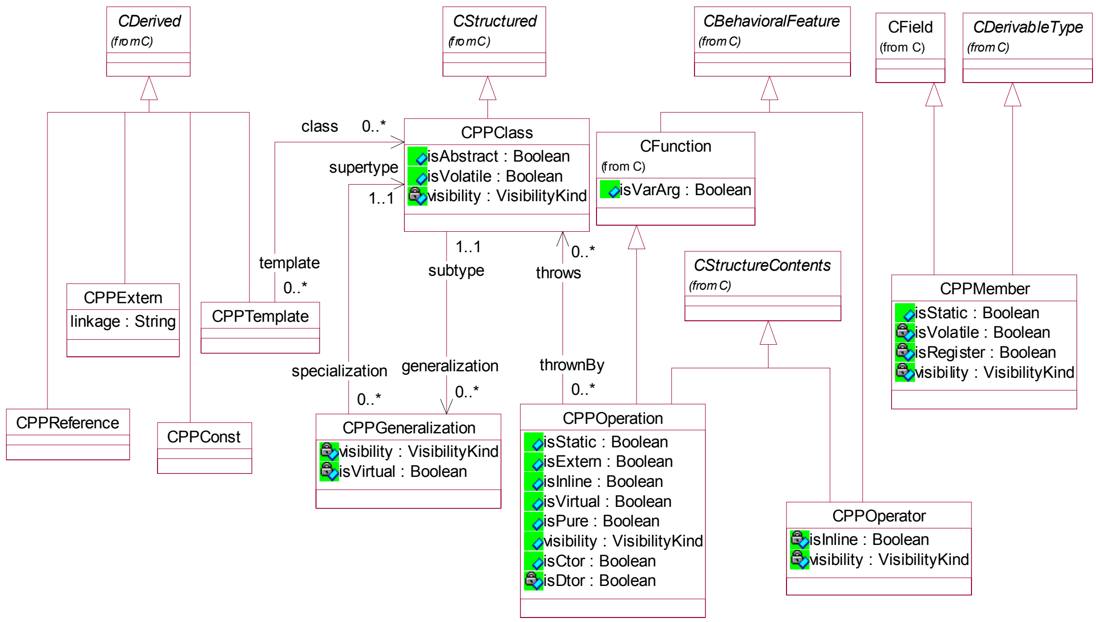



*Figure 14-13* CPP Metamodel

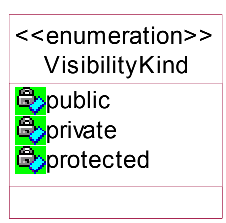



*Figure 14-14* CPP Model Types

## 14.4.1 C++ Metamodel Descriptions

### 14.4.1.1 CPPClass

CPPClass represents the C++ class. The only difference between a C structure and a class is that structure members have public access by default and class members have private access by default. Consequently, you can use the keywords class or struct to define equivalent classes.

### 14.4.1.2 CPPConst

CPPConst represents data declared as a constant.

### 14.4.1.3 CPPExtern

CPPExtern represents a function declared in a C program that is called by the current C++ program. Declaring a function with the keyword 'extern' flags the C++ compiler not to generate an internal name for the function. As a result, functions declared extern may not be overloaded.

### 14.4.1.4 CPPGeneralization

CPPGeneralization represents the different types of generalizations available in a C++ class. Generalizations include associating a class with virtual inheritance.

### 14.4.1.5 CPPMember

CPPMember represents functions and variables that are prototyped and declared in a class definition. CPPMember includes members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, and user-defined types.

### 14.4.1.6 CPPOperation

CPPOperation represents C++ functions. CPPOperation is a specialization of CFunction from the C Metamodel and provides additional features such as static declaration.

### 14.4.1.7 CPPOperator

CPPOperator represents basic operators such as add, subtract, and equals. C++ programmers have the option to override CPPOperators.

### 14.4.1.8 CPPReference

CPPReference represents a reference to an object. References are denoted by an ampersand (&) sign.

### 14.4.1.9 CPPTemplate

CPPTemplate represents a template that must define or declare one of the following:

A class

A function

A static member of a template class

# Non-normative Enterprise Application Interface Metamodels A

The application-domain interface metamodel describes signatures for input and output parameters and return types for enterprise application system domains. IBM's IMS Transaction Message, IMS Message Format Service (MFS), and CICS Basic Mapping Support (BMS) are examples of such metamodels. The payload of these interface metamodels typically carries application data destined for a program of a specific language. Therefore, it is important that these interface metamodels connect to the language metamodels, as shown in Figure 7-7 on page 7-11 in Section 7.3.9, "Physical Representation Model: TDLang Interaction Diagram," on page 7-23. The class in the interface metamodel that represents the signature of a message, associates to a language-independent interface class, TDLangElement, to be able to connect to any language metamodel. From TDLangElement navigations can be done between the Type Descriptor meta model and the language metamodel to perform type conversion, if necessary.

## A.1 IMS Transaction Message Metamodel

IMS OTMA (Open Transaction Manager Access) is a transaction-based, connectionless client/server protocol within an OS/390 sysplex environment. An IMS OTMA transaction message consists of an OTMA prefix, plus message segments for input and output requests. Both input and output message segments contain llzz (i.e., length of the segment and reserved field), and application data. Only the very first input message segment will contain transaction code in front of the application data. IMS transaction application programs can be written in a variety of languages (e.g., COBOL, PL/I, C, Java, etc.); therefore, the application data can be in any one of these languages.

IMS Transaction Message metamodel captures the metadata associated with sending and receiving messages to and from IMS transaction applications. ApplicationData class represents the payload message. Note that the payload message data can be both input and output data parameters. The following figures illustrate the classes that constitute the IMS Transaction Message metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

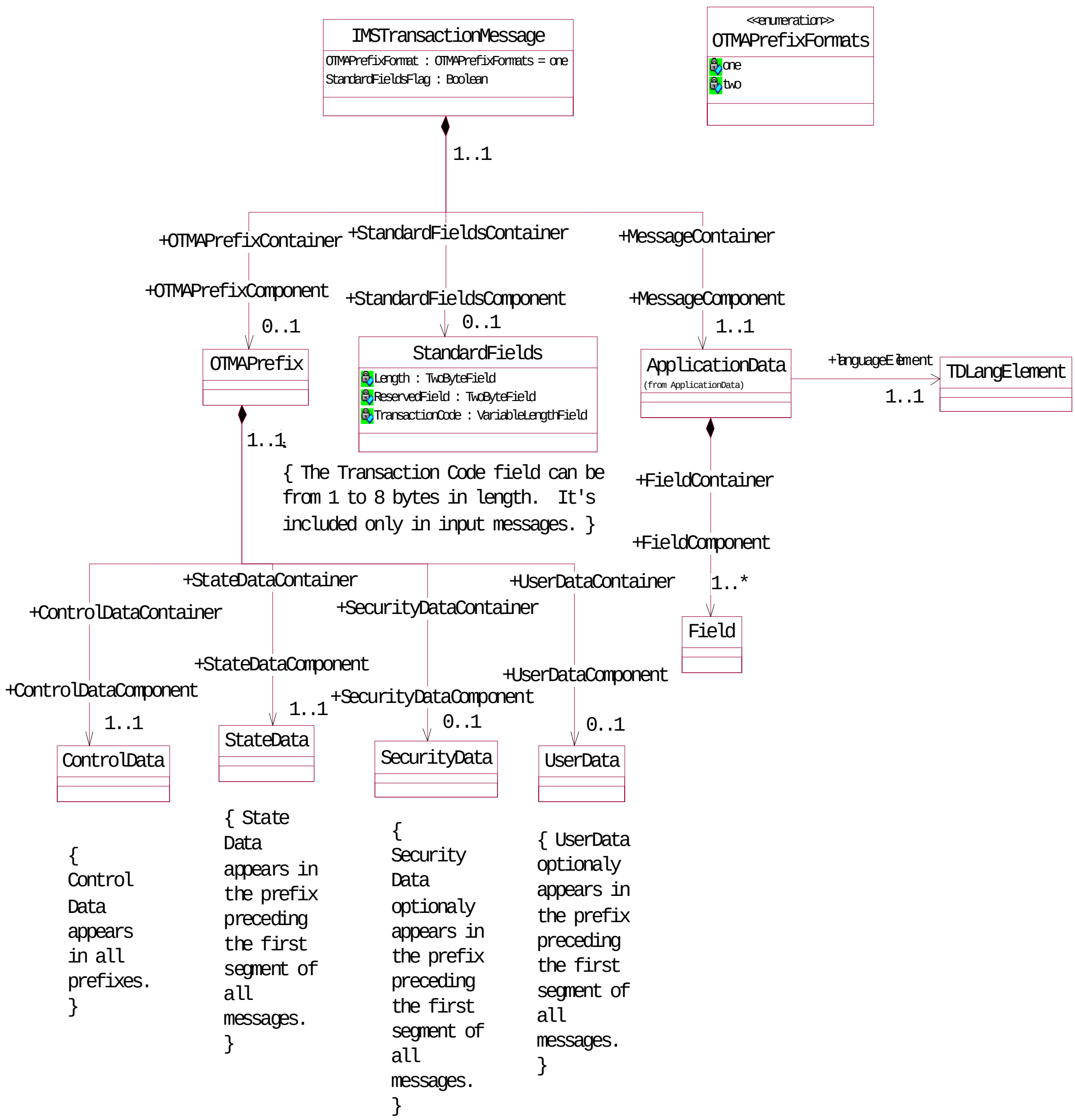



*Figure A-1* IMS Transaction Message Metamodel

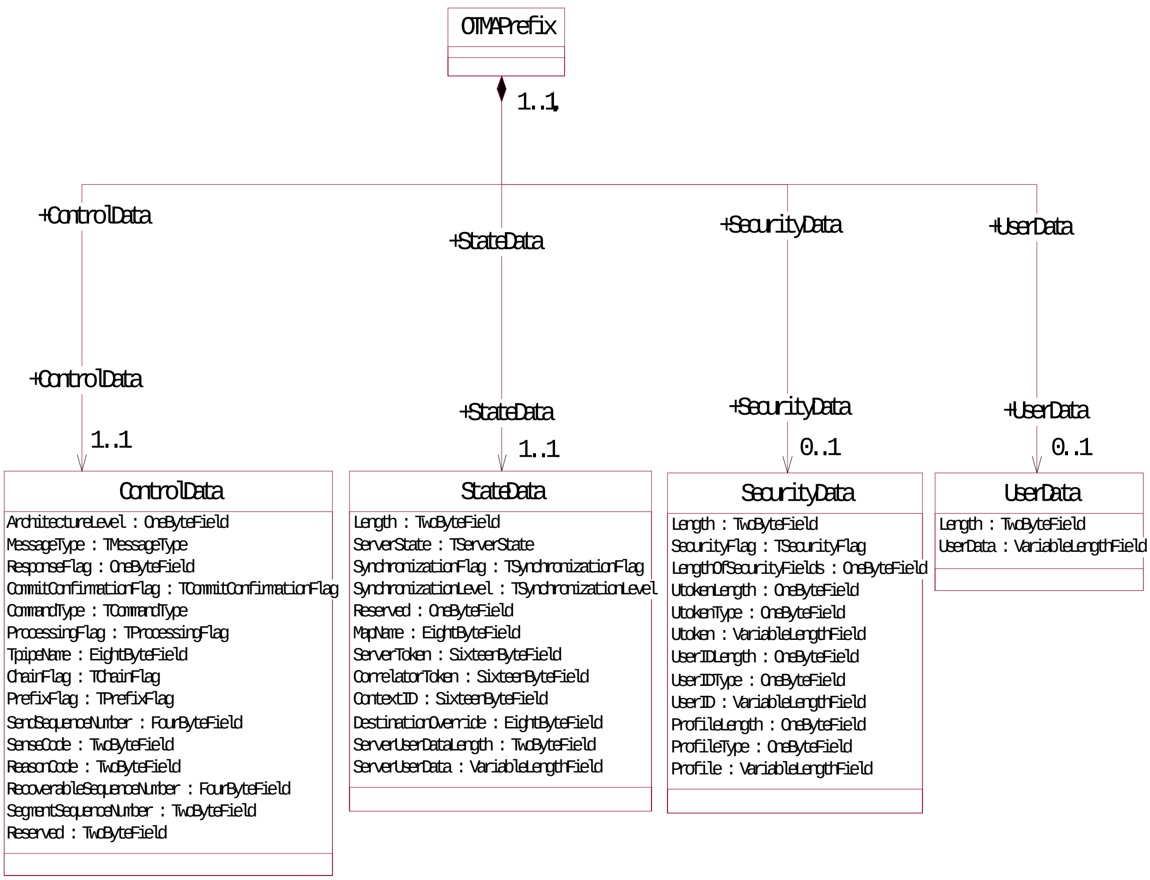



*Figure A-2* IMS Transaction Message Prefix

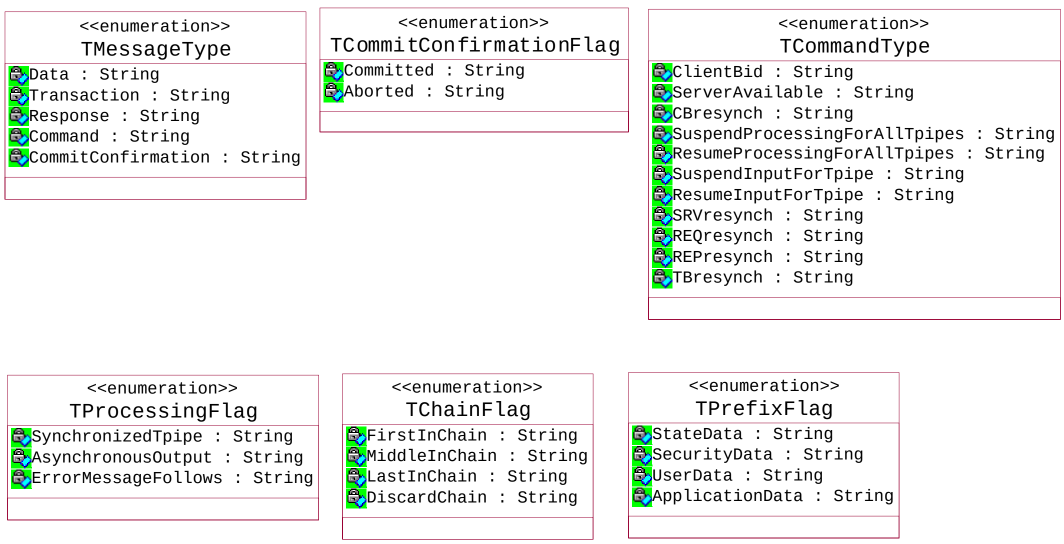



*Figure A-3* OTMA Prefix - Defined Types

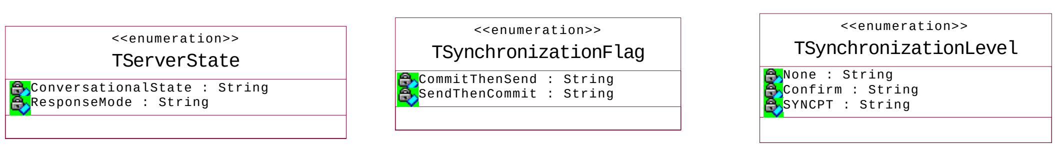



*Figure A-4* OTMA Prefix - State Data Defined Types

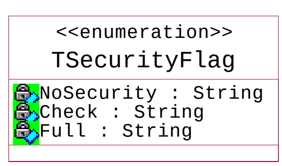



*Figure A-5* OTMA Prefix - Security Data Defined Types


*Figure A-6* IMS Messages Primitive Types

## A.1.1 IMS Transaction Message Metamodel Descriptions

### ApplicationData

The application data class contains all the message data except for LL, ZZ, and the transaction code. ApplicationData contains the signature of an IMS transaction message, which can include inputs, output, and return types. ApplicationData associates with TDLangElement, which provides the linkage to the language specific physical representation of the data that an ApplicationData represents.

**Note –** This model does not capture the notion of message segments. When using this model you have to bear in mind whether the system you are using has any limitations such as a maximum segment size. IMS "gateway" (via OTMA or SNA) must support the capability of breaking the "application data" into IMS message segments.

For instance, if you are sending this XML message directly to the IMS message queue and if the message queue has a 32k limit, then you have to take your XML message and break it up into 32k chunks. The application on IMS will then have to gather up the 32k chunks one by one. IMS new applications that receive XML documents directly, must be capable of receiving XML documents in multiple segments.

For ACK or NAK messages, there is no application data included in the message field.

Each data field, defined in a copybook for the application data, will be associated with type descriptor for data types.

### *ControlData*

ControlData is message-control information. It includes the transaction-pipe name, message type, sequence numbers, flags, and indicators.

ControlData has the following private attributes:

- ArchitectureLevel is an OneByteField.

  Specifies the OTMA architecture level. The client specifies an architecture level, and the server indicates in the response message which architecture level it is using. The architecture levels used by a client and a server must match.

  With IMS Version 6, the only valid value is X'01'. It is mandatory for all messages.

- MessageType is TmessageType.

  Specifies the message type. Every OTMA message must specify a value for the message type. The values are not mutually exclusive. For example, when the server sends an ACK message to a client-submitted transaction, both the transaction and response flags are set.

- ResponseFlag is OneByteField.

  Specifies either that the message is a response message or that a response is requested.

  Acknowledgments to transactions include attributes (for that transaction) in the application-data section of the message prefix only if the transaction specifies Extended Response Requested.

- CommitConfirmationFlag is TcommitConfirmationFlag.

  Specifies the success of a commit request. Sent by the server to the client in a commit-confirmation message. These messages are only applicable for send-then-commit transactions, and are not affected by the synchronization-level flag in the state-data section of the message prefix.

- CommandType is TcommandType.

  Specifies the OTMA protocol command type.

  IMS commands are specified in the application-data section of the message.

- ProcessingFlag is TprocessingFlag.

  Specifies options by which a client or a server can control message processing.

- TpipeName is EightByteField.

  Specifies the transaction-pipe name. For IMS, this name is used to override the LTERM name on the I/O PCB. This field is applicable for all transaction, data, and commit-confirmation message types. It is also applicable for certain response and command message types.

- ChainFlag is TchainFlag.

Specifies how many segments are in the message. This flag is applicable to transaction and data message types, and it is mandatory for multi-segment messages.

- PrefixFlag is TprefixFlag.

  Specifies the sections of the message prefix that are attached to the OTMA message. Every message must have the message-control information section, but any combination of other sections can be sent with an OTMA message.

- SendSequenceNumber is FourByteField.

  Specifies the sequence number for a transaction pipe. This sequence number is updated by the client and server when sending message or transactions.

  Recommendation: Increment the number separately for each transaction pipe.

  This number can also be used to match an ACK or NAK message with the specific message being acknowledged.

- SenseCode is TwoByteField.

  Specifies the sense code that accompanies a NAK message.

- ReasonCode is TwoByteField.

  Specifies the reason code that accompanies a NAK message. This code can further qualify a sense code.

- RecoverableSequenceNumber is FourByteField.

  Specifies the recoverable sequence number for a transaction pipe. Incremented each time a recoverable message is sent using a synchronized transaction pipe. Both the client and the server increment their recoverable send-sequence numbers and maintain them separately from the send-sequence number.

- SegmentSequenceNumber is TwoByteField.

  Specifies the sequence number for a segment of a multi-segment message. This number must be updated for each segment, because messages are not necessarily delivered sequentially by XCF.

  This number must have a value of 0 (zero) if the message has only one segment.

- Reserved is a TwoByteField.

### *IMSTransactionMessage*

IMSTransactionMessage is the base class of the IMS transaction message metamodel which includes the following IMS messages scenarios:

- IMS OTMA messages with the OTMA prefix
- IMS OTMA messages without the OTMA prefix
- IMS basic messages to be sent to the application program directly

### *OTMAPrefix*

An IMS OTMA prefix can appear either before all message segments, or only before the first segment of the message.

However, the OTMA prefix is optional. If it is not specified, the IMS gateway will build a default one for the request.

### *OTMAPrefixFormats*

OTMAPrefixFormats has the following two types:

- Format "one": a prefix appears before all message segments.
- Format "two": a prefix appears only before the first message segment.

### *SecurityData*

SecurityData includes the user ID, user token, and security flags.

The security-data section is mandatory for every transaction, and can be present for OTMA command messages.

SecurityData has the following private attributes:

- Length is TwoByteField.

  Specifies the length of the security data section of the message prefix, including the length field.

- SecurityFlag is TsecurityFlag.

  Specifies the type of security checking to be performed. It is assumed that the user ID and password are already verified.

- LengthOfSecurityFields is OneByteField.

  Specifies the length of the security data fields: User ID, Profile, and Utoken. These three fields can appear in any order, or they can be omitted. Each has the following structure: Length field, then Field type, then Data field. The actual length of the User ID or Profile should not be less than the value specified for the length of each field.

  Length can be 0.

- UtokenLength is OneByteField.

  Specifies the length of the user token. Length does not include length field itself.

- UtokenType is OneByteField.

  Specifies that this field contains a user token. (Value X'00').

- Utoken is VariableLengthField.

Specifies the user token. The user ID and profile are used to create the user token. The user token is passed along to the IMS dependent region.

If the client has already called FACF, it should pass the Utoken with field type X'00' so that RACF is not called again. Utoken is a variable length, from 1 to 80 bytes.

- UserIDLength is OneByteField.

  Specifies the length of the user ID. Length does not include length field itself.

- UserIDType is OneByteField.

  Specifies that this field contains a user ID. (Value X'02').

- UserID is VariableLengthField.

  Specifies the actual user ID. UserID is a variable length, from 1 to 10 bytes.

- ProfileLength is OneByteField.

  Specifies the length of the profile. Length does not include length field itself.

- ProfileType is OneByteField.

  Specifies that this field contains a profile. (Value X'03').

- Profile is VariableLengthField.

  Specifies the system authorization facility (SAF) profile. For RACF, this is the group name. Profile is a variable length, from 1 to 10 bytes.

### *StandardFields*

StandardFields consist of LL, ZZ, and transaction code. Transaction code appears with first segment of input messages only, and it comes after LL (length) and ZZ (reserved field). The transaction code field can be from 1 to 8 bytes in length.

StandardFields are not included in the following scenarios:

- Sending XML documents directly to the IMS transaction application programs.
- ACK or NAK messages to IMS applications.

### *StateData*

StateData includes a destination override, map name, synchronization level, commit mode, tokens, and server state.

StateData has the following private attributes:

- Length is a of type TwoByteField.
- ServerState is of type ServerState. It specifies the mode in which the transaction is running.

- SynchronizationFlag is of type TsynchronizationFlag. It specifies the commit mode of the transaction. This flag controls and synchronizes the flow of data between the client and server.
- SynchronizationLevel is of type TsynchronizationLevel. It specifies the transaction synchronization level, the way in which the client and server transaction program (for example, IMS application program) interacts with program output messages.

The default is Confirm. IMS always requests a response when sending commit-then-send output to a client.

- Reserved is OneByteField.
- MapName is EightByteField.

  Specifies the formatting map used by the server to map output data streams (for example, 3270 data streams). Although OTMA does not provide MFS support, you can use the map name to define the output data stream. The name is an 8-byte MOD name that is placed in the I/O PCB. IMS replaces this field in the prefix with the map name in the I/O PCB when the message is inserted. The map name is optional.

- ServerToken is SixteenByteField.

  Specifies the server name. The Server Token must be returned by the client to the server on response messages (ACKs or NAKs). For conversational transactions, the Server Token must also be returned by the client on subsequent conversational input.

- CorrelatorToken is SixteenByteField.

  Specifies a client token to correlate input with output. This token is optional and is not used by the server.

  Recommendation: Clients should use this token to help manage their transactions.

- ContextID is SixteenByteField.

  Specifies the RRS/MVS token that is used with SYNCLVL=02 and protected conversations.

- DestinationOverride is EightByteField.

  Specifies an LTERM name used to override the LTERM name in the IMS application program's I/O PCB. This override is used if the client does not want to override the LTERM name in the I/O PCB with the transaction-pipe name.

  This optional override is not used if it begins with a blank.

- ServerUserDataLength is TwoByteField.

  Specifies the length of the server user data, if any. The maximum length of the server use data is 256 bytes.

- ServerUserData is VariableLengthField.

  Specifies any data needed by the server. If included in a transaction message by the client, it is returned by the server in the output data messages.

### *TChainFlag*

TchainFlag has the following private attributes:

- FirstInChain (value X'80') specifies the first segment in a chain of segments, which comprise a multi-segment message. Subsequent segments of the message only need the message-control information section of the message prefix. Other applicable prefix segments (for example, those specified by the client on the transaction message) are sent only with the first segment (with the first-in-chain flag set).

  If the OTMA message has only one segment, the last-in-chain flag should also be set.

- MiddleInChain (value X'40') specifies a segment that is neither first nor last in a chain of segments that comprise a multi-segment message. These segments only need the message-control information section of the message prefix.

  Restriction: Because the client and server tokens are in the state-data section of the message prefix, they cannot be used to correlate and combine segmented messages. The transaction-pipe name and send-sequence numbers can be used for this purpose; they are in the message-control information section of the message prefix for each segment.

- LastInChain (value X'20') specifies the last segment of a multi-segment message.
- DiscardChain (value X'10') specifies that the entire chain of a multi-segment message is to be discarded. The last-in-chain flag must also be set.

### *TCommandType*

TcommandType has the following private attributes:

- ClientBid (value X'04') specifies the first message a client sends to the OTMA server. This command must also set the response-requested flag and the security flag in the message-control information section of the message prefix. The appropriate stat-data fields (for example, Member Name) must also be set.
- The security-data prefix must specify a Utoken field so the OTMA server can validate the client's authority to act as an OTMA client.

  Because the server can respond to the client-bid request, this message should not be sent until the client is ready to start accepting data messages.

- ServerAvailable (value X'08') specifies the first message the server sends to a client. It is sent when the server has connected to the XCF group before the client has connected. The client replies to the server Available message with a client-bid request. The appropriate state data fields (for example, Member Name) must also be set.

  If the client connects first, it is notified by XCF when the server connects, and begins processing with a client-bid request.

- CBresynch (value X'0C') specifies a client-bid message with a request by the client for resynchronization. This command is optional and causes the server to send an SRVresynch message to the client. The CBresynch command is the first message that a client sends to the OTMA server when it attempts to resynchronize with IMS and existing synchronized Tpipes exist for the client. Other than the CBresynch message indicator in the message prefix, the information required for the message prefix should be identical to the client-bid command.

  If IMS receives a client-bid request for them client and IMS is aware of existing synchronized Tpipes, IMS issues informational message DFS2394I to the MTO. IMS resets the recoverable send- or receive- sequence numbers to 0 (zero) for all the synchronized Tpipes.

- SuspendProcessingForAllTpipes (value X'14') specifies that the server is suspending all message activity with the client. All subsequent data input receives a NAK message from the server. Similarly, the client should send a NAK message for any subsequent server messages. If a client wishes to suspend processing for a particular transaction pipe, it must submit a /STOP TPIPE command as an OTMA message.
- ResumeProcessingForAllTpipes (value X'18') specifies that the server is resuming message activity with the client. If a client wishes to resume processing for a particular transaction pipe that has been stopped, it must submit a /START TPIPE command as an OTMA message.
- SuspendInputForTpipe (value X'1C') specifies that the server is overloaded and is temporarily suspending input for the transaction pipe. All subsequent client input receive NAK messages for the transaction pipe specified in the message-control information section of the message prefix. A response is not requested for this command.
- This architected command is also sent by IMS when the master terminal operator enters a /STOP TPIPE command.
- ResumeInputForTpipe (value X'20') specifies that the server is ready to resume client input following an earlier Suspend Input for Tpipe command. A response is not requested for this command.

  This command is also sent by IMS when the IMS master terminal operator issues a /START TPIPE command.

- SRVresynch (value X'2C') specifies the server's response to a client's CBresynch command. This command specifies the states of synchronized transaction pipes within the server (the send- and receive-sequence numbers).

  This command is sent as a single message (with single or multiple segments), and an ACK is requested.

- REQresynch (value X'30') specifies the send-sequence number and the receive sequence for a particular Tpipe. REQresynch is send from IMS to a client.
- REPresynch (value X'34') specifies the client's desired state information for a Tpipe. A client sends the REPresynch command to IMS in response to the REQresynch command received from IMS.

- TBresynch (value X'38') specifies that the client is ready to receive the REQresynch command from IMS.

### TCommitConfirmationFlag

TcommitConfirmationFlag has the following private attributes:

- Committed (value X'80') specifies that the server committed successfully.
- Aborted (value X'40') specifies that the server aborted the commit.

### TMessageType

TmessageType has the following private attributes:

- Data (value X'80') specifies server output data sent to the client. If the client specifies synchronization level Confirm in the state-data section of the message prefix, the server also sets Response Requested for the response flag. If the client does not specify a synchronization level, the server uses the default, Confirm.
- Transaction (value X'40') specifies client input data to the server.
- Response (value X'20') specifies that the message is a response message, and is only set if the message for which this message is the response specified Response Requested for the response flag. If this flag is set, the response flag specifies either ACK or NAK.
- The send-sequence numbers must match for the original data message and the response message. Chained transaction input messages to the server must always request a response before the next transaction (for a particular transaction pipe) is sent.
- Command (value X'10') specifies an OTMA protocol command. OTMA commands must always specify Response Requested for the Response flag.
- CommitConfirmation (value X'08') specifies that commit is complete. This is sent by the server when a sync point has completed, and is only applicable for send-then-commit transactions. The commit-confirmation flag is also set.

### TPrefixFlag

TPrefixFlag has the following attributes:

- StateData (value X'80') specifies that the message includes the state-data section of the message prefix.
- SecurityData (value X'40') specifies that the message includes the security-data section of the message prefix.
- UserData (value X'20') specifies that the message includes the user-data section of the message prefix.
- ApplicationData (value X'10') specifies that the message includes the application-data section of the message prefix.

### *TProcessingFlag*

TprocessingFlag has the following private attributes:

- SynchronizedTpipe (value X'40') specifies that the transaction pipe is to be synchronized. Allows the client to resynchronize a transaction pipe if there is a failure. Only valid for commit-then-send transactions.

  This flag causes input and output sequence numbers to be maintained for the transaction pipe. All transactions routed through the transaction pipe must specify this flag consistently (either on or off).

- AsynchronousOutput (value X'20') specifies that the server is sending unsolicited queued output to the client. This can occur when IMS inserts a message to an alternate PCB. Certain IMS commands, when submitted as commit-then-send, can cause IMS to send the output to a client with this flag set. In this case, the OTMA prefixes contain no identifying information that the client can use to correlate the output to the originating command message. These command output data messages simply identify the transaction-pipe name. IMS can also send some unsolicited error messages with only the transaction-pipe name.

- ErrorMessageFollows (value X'10') specifies that an error message follows this message. This flag is set for NAK messages from the server. An additional error message is then sent to the client.

  The asynchronous-output flag is not set in the error data message, because the output is not generated by an IMS application.

### *TResponseFlag*

TResponseFlag has the following private attributes:

- ACK (value X'80') specifies a positive acknowledgment.
- NAK (value X'40') specifies a negative acknowledgment.
- ResponseRequested (value X'20') specifies that a response is requested for this message. This can be set for message types of Data, Transaction, or Command.
- When sending send-then-commit IMS command output, IMS does not request an ACK regardless of the synchronization level.
- ExtendedResponseRequested (value X'10') specifies that an extended response is requested for this message. Can be set by a client only for transactions (or for transactions that specify an IMS command instead of a transaction code).
- If this flag is set for a transaction, IMS returns the architected attributes for that transaction in the application-data section of the ACK message.
- If this flag is set for a command, IMS returns the architected attributes in the application-data section of the ACK message. This flag can be set for the IMS commands /DISPLAY TRANSACTION and /DISPLAY TRANSACTION ALL.

### *TSecurityFlag*

TSecurityFlag has the following attributes:

- NoSecurity (value X'N') specifies that no security checking is to be done.
- Check (value X'C') specifies that transaction and command security checking is to be performed.
- Full (value X'F') specifies that transaction, command, and MPP region security checking is to be performed.

### *TServerState*

TServerState has the following private attributes:

- ConversationalState (value X'80') specifies a conversational mode transaction. The server sets this state when processing a conversational-mode transaction. This state is also set by the client when sending subsequent IMS conversational data messages to IMS.
- ResponseMode (value X'40') specifies a response-mode transaction. Set by the server when processing a response-mode transaction.

This state has little significance for an OTMA server, because OTMA does not use sessions or terminals.

### *TSynchronizationFlag*

TSynchronizationFlag has the following private attributes:

- CommitThenSend (value X'40') specifies a commit-then-send transaction. The server commits output before sending it; for example, IMS inserts the output to the IMS message queue.
- SendThenCommit (value X'20') specifies a send-then-commit transaction. The server sends output to the client before committing it.

### *TSynchronizationLevel*

TSynchronizationLevel has the following private attributes:

- None (value X'00') specifies that no synchronization is requested. The server application program does not request an ACK message when it sends output to a client.
- None is only valid for send-then-commit transactions.
- Confirm (value X'01') specifies that synchronization is requested. The server sends transaction output with the response flag set to Response Requested in the message-control information section of the message prefix.

  Confirm can be used for either commit-then-send or send-then-commit transactions.

- SYNCPT (value X'02') specifies that the programs participate in coordinated commit processing on resources updated during the conversion under the RRS/MVS recovery platform. A conversation with this level is also called a protected conversation.

### UserData

UserData includes any special information needed by the client. The user-data section is variable length and follows the security-data section of the message prefix. It can contain any data.

UserData has the following attributes:

- Length is a TwoByteField.

  Specifies the length of the user-data section of the message prefix, including the length field. The maximum length of the user data is 1024 bytes.

- UserData is a VariableLengthField.

  Specifies the optional user data. This data is managed by the client, and can be created and updated using the DFSYDRU0 exit routine. The server returns this section unchanged to the client as the first segment of any output messages.

## A.2 IMS MFS Metamodel

Today there are many IMS application programs which run crucial business processes. Many of these IMS programs are based on IMS's message format service (MFS). MFS is a facility of the IMS Transaction Manager environment that formats messages to and from terminal devices. As these business processes are updated to exploit new business-to-business (B2B) technologies, there is a requirement for an easy and effective method of upgrading MFS applications with e-business capabilities. What is needed is the ability to send and receive IMS transaction messages, including MFS messages, as XML documents.

The MFS language utility processes MFS source, generates IMS control blocks, in a proprietary format, known as Message Input/Output Descriptors (MID/MOD) and Device Input/Output Format (DIF/DOF), and places them in an IMS Format Library. MFS supports several terminal types, including 3270s and VTAM LU1s using SCS, it was designed so that the IMS application programs using MFS do not themselves have to deal with any device-specific characteristics in the input or output messages. Because MFS provides headers, page numbers, operator instructions, and other literals to the device, the application's input and output messages can be built without having to pass these format literals. MFS identifies all fields in the message response and formats these responses according to the specific device type that is the target for the response. This allows application programmers to concentrate their efforts on the business logic of the program.

Because the IMS application program I/O data structures do not fully describe the end user interaction with these existing MFS applications, a way is needed to deal with the information that is buried within various MFS statements. Important examples of this kind of information are 3270 screen attribute bytes and PFKey input data. PFKeys can have significant semantic meaning for an application; it can even be used to initiate transactions. Many IMS application programs are passed PFKey data in input messages, but no application logic is required to recognize that a certain PFkey was pressed and therefore must be inserted into the input message. This is because, at runtime, it is the MFS online processing and not the application that places the literal that corresponds to the PFKey pressed into the appropriate field in the input message.

The IMS MFS metamodel, modeled from the MFS source, captures certain services or functions currently provided by MFS. Examples of such services or functions are PF keys, logical pages, predefined literals, and attribute bytes.

Note that the MFS metamodel supports the following device types:

- 3270 and 3270-An
- 3270P

The following device types are not supported:

- 2740 or 2741
- 3600 or 4700
- FIN
- FIDS, FIDS3, FIDS4 or FIDS7
- FIJP, FIPB or FIFP
- SCS1
- SCS2
- DPM-An
- DPM-Bn

The MFS metamodel does not support the following MFS statements:

- EJECT
- PD
- PDB
- PDBEND
- PPAGE (partial support, see DFLD)
- PRINT
- RCD
- SPACE
- TITLE

MFSMessageField identifies the signature of an IMS FMS message, which can include both inputs and outputs. MFSMessageField associates with TDLangElement, which provides the linkage to the language and platform specific representations of the data.

The following figures illustrate the classes that constitute the IMS MFS metamodel and show how the classes relate to each other.  Following the diagrams is a brief explanation of what each class represents.

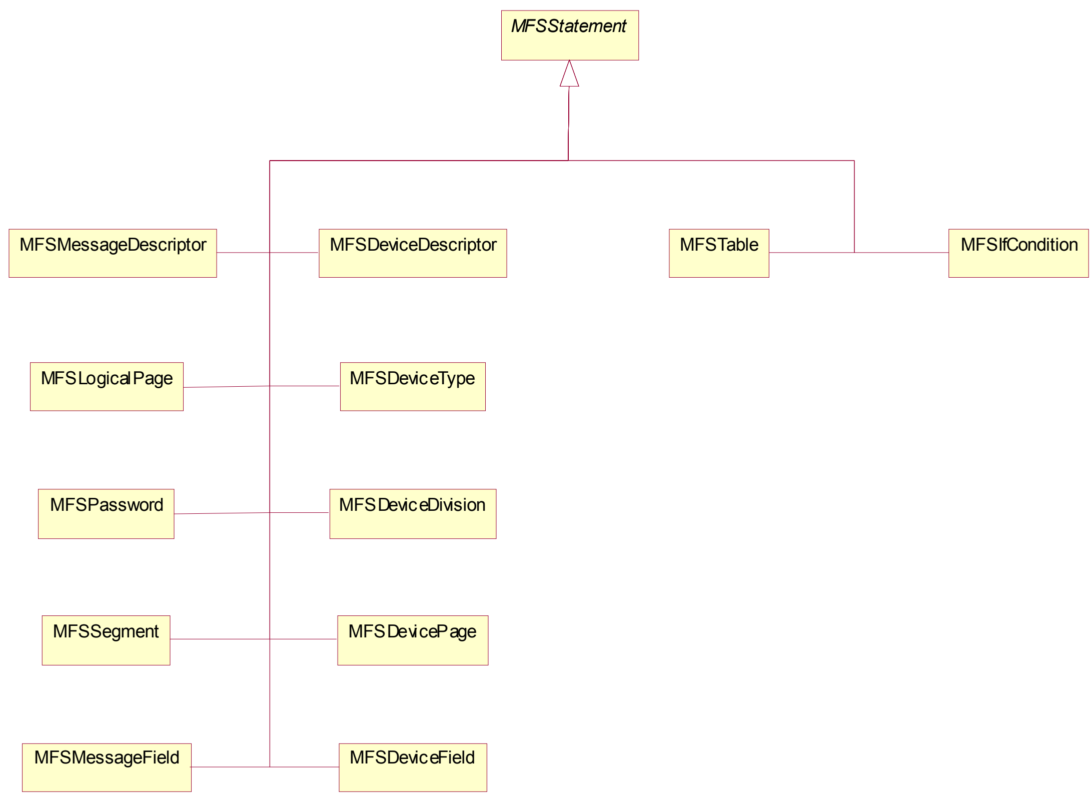



*Figure A-7*    MFS Inheritance View

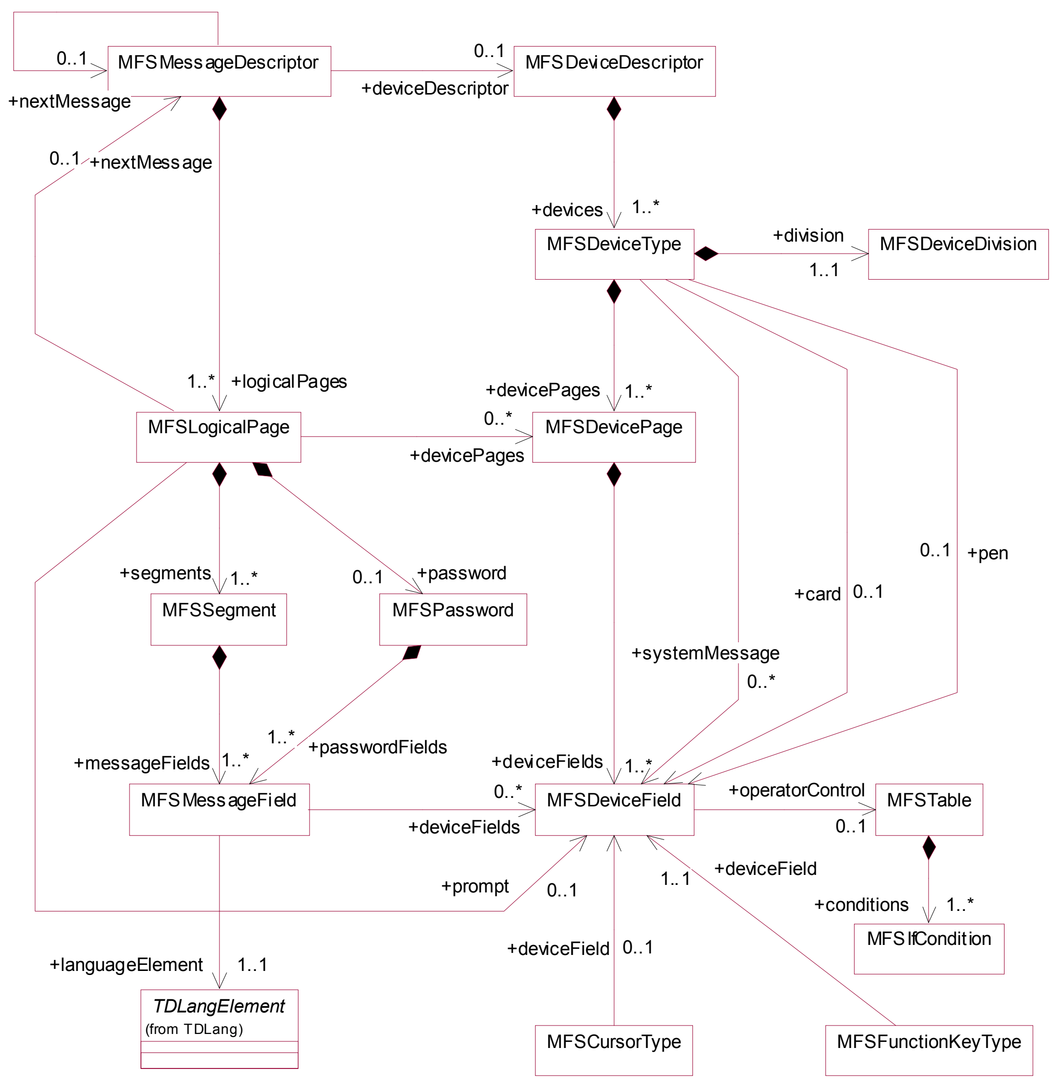



*Figure A-8* MFS Relationship View

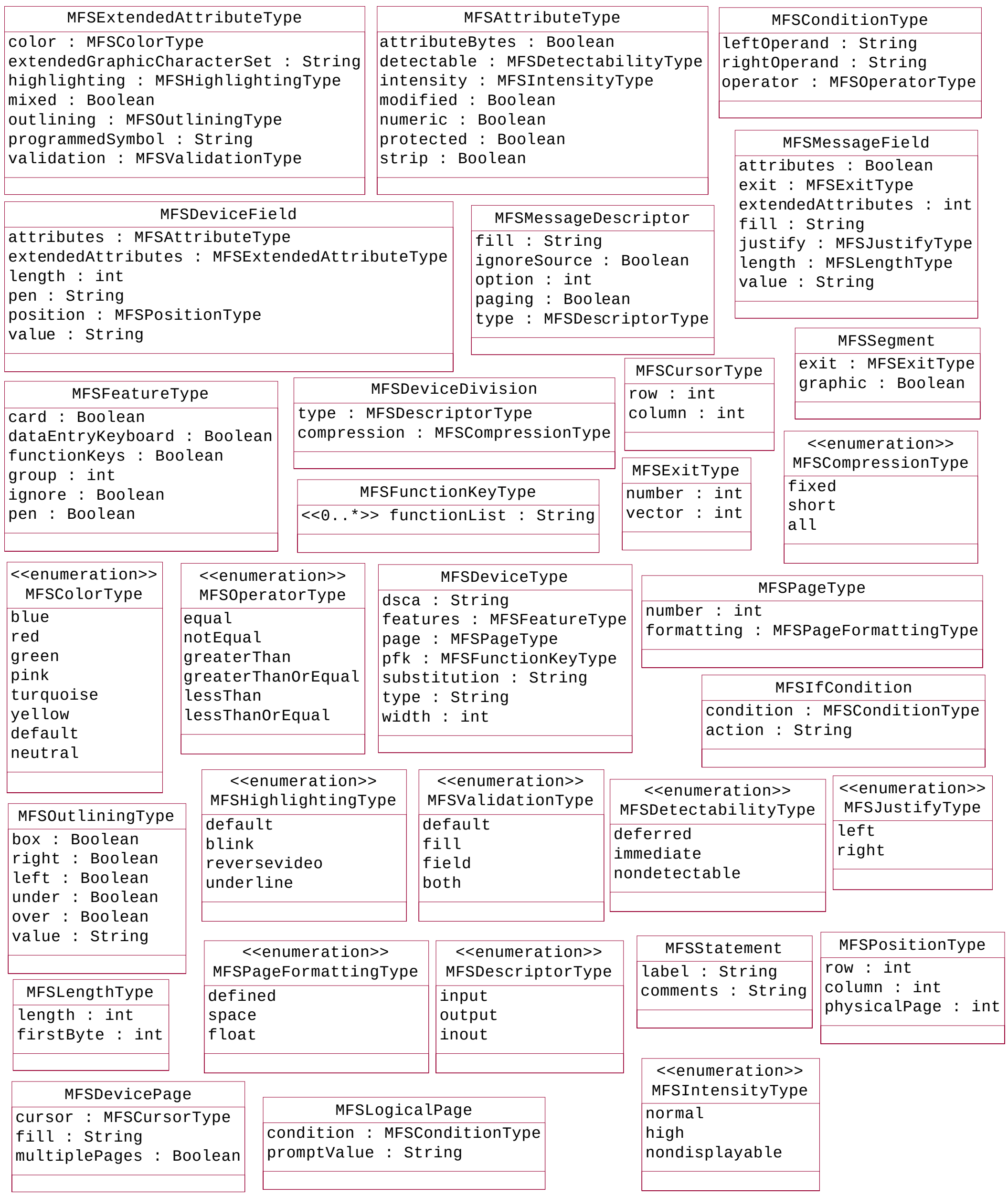



*Figure A-9* MFS Attribute View

### *A.2.1 IMS MFS Metamodel Descriptions*

#### *MFSDeviceDescriptor*

This class encapsulates the MFS "FMT" statement.

The FMT statement initiates and names a format definition that includes one or more device formats differing only in the device type and features specified in the DEV statement. Each device format included in the format definition specifies the layout for data sent to or received from a device or a remote program. All attributes are supported.

#### *MFSDeviceDivision*

This class encapsulates the MFS "DIV" statement.

The DIV statement defines device formats within a DIF or DOF. The formats are identified as input, output, or both input and output, and can consist of multiple physical pages. Only one DIV statement per DEV is allowed.

The MFS metamodel does not support the following DIV attributes:

- RCDCTL
- HDRCTL
- OPTIONS
- OFTAB
- DPN
- PRN
- RDPN
- RPRN

***type : MFSDescriptorType***

*TYPE attribute*

Describes an input only format (INPUT), an output only format (OUTPUT), or both (INOUT).

If DIV TYPE=OUTPUT or TYPE=INPUT is specified, certain DEV statement keywords are applicable.

***compression : MFSCompressionType***

*COMPR attribute*

Requests MFS to remove trailing blanks from short fields, fixed-length fields, or all fields presented by the application program.

### *MFSDeviceField*

This class encapsulates the MFS “DFLD” statement.

The DFLD statement defines a field within a device format, which is read from or written to a terminal or remote program. Only those areas, which are of interest to the IMS or remote application program should be defined. Null space in the format does not need to be defined. The SLD attribute is not supported.

#### ***attributes : MFSAttributeType***

*ATTR attribute*

#### ***extendedAttributes : MFSExtendedAttributeType***

*EATTR attribute*

#### ***length : int***

*LTH attribute*

Specifies the length of the field. This operand should be omitted if 'literal' is specified in the positional parameter, in which case the length of literal is used as the field length. Unpredictable formatting output can occur if this operand is used in conjunction with a 'literal' and the two lengths are different. The specified LTH= cannot exceed the physical page size of the device.

The maximum allowable length for all devices except 3270, 3604 display, and DPM with RCDCT=NOSPAN is 8000 characters. For 3270 displays, the maximum length is one less than screen size. For example, for a 480-character display, the maximum length is 479 characters. A length of 0 must not be specified. If SCA and LTH= are both specified, LTH must be 2.

POS= and LTH= do not include the attribute character position reserved for a 3270 display device or a DFLD with ATTR=YES specified. The inclusion of this byte in the design of display/printer formats is necessary because it occupies the screen/printed page position preceding each displayed/printed field even though it is not accessible by an application program.

When defining DFLDs for 3270 printers, a hardware ATTRIBUTE character is not used. Therefore, fields must be defined with a juxtaposition that does not allow for the attribute character unless ATTR=YES is specified. However, for printers defined as 3270P the last column of a print line (based on FEAT=, WIDTH=, or the device default width) cannot be used. The last column of the line is reserved for carriage control operations performed by IMS. Thus, if the print line specifies 120 (FEAT=120) and the DFLD specifies POS=(1,1),LTH=120 then 119 characters are printed on line 1 and one character on line 2.

Detectable fields (DET or IDET) must include four positions in POS and LTH for a 1-byte detection designator character and 3 pad characters, unless the detectable field is the last field on a display line, in which case only one position for the detection designator character is required. The detection designator character must precede field data, and pad characters (if required) follow field data. Detection designator and required pad characters must be supplied by the application program or MFLD literal with the field data. Pad characters can also be required in the preceding field on the device.

### *pen : String*

*PEN attribute*

Specifies a literal to be selected or an operator control function to be performed when this field is detected. If (1) 'literal' is specified, (2) the field is defined as immediately detectable (ATTR= operand), and (3) contains the null or space designator character, the specified literal is placed in the field referred to by the PEN operand of the preceding DEV statement when the field is detected (if no other device fields are modified). If another field on the device is modified, a question mark (?) is provided instead of the literal. Literal length must not exceed 256 bytes.

If (1) a control function is specified, (2) the field is defined as immediately detectable (ATTR= operand), and (3) contains the null or space designator character, the specified control function is performed when the field is detected and no other device fields are modified. If another field on the device is modified, a question mark (?) is provided and the function is not performed. Control functions that can be specified are:

- NEXTPP--PAGE ADVANCE specifies a request for the next physical page in the current output message. If no output message is in progress, no explicit response is made.
- NEXTMSG--MESSAGE ADVANCE specifies a request to dequeue the output message in progress (if any) and to send the next output message in the queue (if any).
- NEXTMSGP--MESSAGE ADVANCE PROTECT specifies a request to dequeue the output message in progress (if any), and send the next output message or return an information message indicating that no next message exists.
- NEXTLP--NEXT LOGICAL PAGE specifies a request for the next logical page of the current message.
- ENDMPPI--END MULTIPLE PAGE INPUT specifies the end of a multiple physical page input message.
- ENDMPPI is valid only if data has been received and will not terminate multiple page input (MPPI) in the absence of data entry.

### *position : MFSPositionType*

*POS attribute*

Defines the first data position of this field in terms of line (lll), column (ccc), and physical page (pp) of the display format. If pp is omitted, 1 is assumed.

For DEV TYPE=3270, 3270-An, or 3270P:

- lll,ccc,pp specifies the line, column, and optionally, the physical page number for an output field. lll, ccc, and pp must be greater than or equal to 1.
- For 3270 displays, POS=(1,1) must not be specified. Fields must not be defined such that they wrap from the bottom to the top.

Restriction: On some models of 3270s, the display screen cannot be copied when a field starting on line 1, column 2, has both alphabetic and protect attributes.

#### *value : String*

The default value of the device field.

### *MFSDevicePage*

This class encapsulates the MFS "DPAGE" statement.

The DPAGE statement defines a logical page of a device format. This statement can be omitted if none of the message descriptors referring to this device format (FMT) contains LPAGE statements and if no specific device option is required. It is implied if not present.

The MFS metamodel does not support the following DPAGE attributes:

- ACTVPID
- COND
- OFTAB
- ORIGIN
- PD
- SELECT

#### *cursor : MFSCursorType*

*CURSOR attribute*

Specifies the position of the cursor on a physical page. Multiple cursor positions may be required if a logical page or message consists of multiple physical pages. The value lll specifies line number, ccc specifies column; both lll and ccc must be greater than or equal to 1. The cursor position must either be on a defined field or defaulted. The default lll,ccc value for 3270 displays is 1,2. For Finance display components, if no cursor position is specified, MFS will not position the cursor--the cursor is normally placed at the end of the output data on the device. For Finance display components, all cursor positioning is absolute, regardless of the ORIGIN= parameter specified.

The dfld parameter provides a method for supplying the application program with cursor information on input and allowing the application program to specify cursor position on output.

Recommendation: Use the cursor attribute facility (specify ATTR=YES in the MFLD statement) for output cursor positioning.

The dfld parameter specifies the name of a field containing the cursor position. This name may be referenced by an MFLD statement and must not be used as the label of a DFLD statement in this DEV definition. The format of this field is two binary halfwords containing line and column number, respectively. When this field is referred to by a message input descriptor, it will contain the cursor position at message entry. If referred to by a message output descriptor, the application program places the desired cursor position into this field as two binary halfwords containing line and column, respectively. Binary zeros in the named field cause the specified lll,ccc to be used for cursor positioning during output. During input, binary zeros in this field indicate that the cursor position is not defined. The input MFLD referring to this dfld should be defined within a segment with GRAPHIC=NO specified or should use EXIT=(0,2) to convert the binary numbers to decimal.

#### *fill : String*

*FILL attribute*

Specifies a fill character for output device fields. Default value for all device types except the 3270 display is X'40'; default for the 3270 display is PT. For 3270 output when EGCS fields are present, only FILL=PT or FILL=NULL should be specified. A FILL=PT erases an output field (either a 1- or 2-byte field) only when data is sent to the field, and thus does not erase the DFLD if the application program message omits the MFLD.

- NONE must be specified if the fill character from the message output descriptor is to be used to fill the device fields.
- X'hh' character whose hexadecimal representation is 'hh' will be used to fill the device fields.
- C'c' character 'c' will be used to fill the device fields.
- NULL specifies that fields are not to be filled. For devices other than the 3270 display, 'compacted lines' are produced when message data does not fill the device fields.
- PT specifies that output fields that do not fill the device field (DFLD) are followed by a program tab character to erase data previously in the field; otherwise, this operation is identical to FILL=NULL.

For 3270 display devices, any specification with a value less than X'3F' is changed to X'00' for control characters or to X'40' for other non-graphic characters.

#### *multiplePages : Boolean*

*MULT attribute*

Specifies that multiple physical page input messages will be allowed for this DPAGE.

### *MFSDeviceType*

This class encapsulates the MFS "DEV" statement.

The DEV statement defines device characteristics for a specific device or data formats for a specific device type. The DFLD statements following this DEV statement are mapped using the characteristics specified until the next DEV or FMTEND statement is encountered.

The MFS metamodel does not support the following DEV attributes:

- ERASE
- FTAB
- FORMS
- HT
- HTAB
- LDEL
- MODE
- SLD
- VERSID
- VT
- VTAB

### *card : 0..1 MFSDeviceField*

*CARD attribute*

Defines the input field name to receive operator identification card data when that data is entered. This name can be referenced by an MFLD statement and must not be used as the label of a DFLD statement within this DEV definition. This operand is valid only if a 3270 display is specified. If FEAT=NOCD is specified for a 3270 display, it is changed to CARD. All control characters are removed from magnetic card input before the data is presented to the input MFLD that refers to this card field name.

For 3270 displays, an unprotected field large enough to contain the magnetic card data and control characters must be defined through a DFLD statement. Position the cursor to this field and insert the card in the reader to enter card information. The card data is logically associated with the CARD= field name, not the name used in the DFLD statement.

### *dsca : String*

*DSCA attribute*

Specifies a default system control area (DSCA) for output messages using this device format. The DSCA supersedes any SCA specified in a message output descriptor if there are conflicting specifications. Normally, the functions specified in both SCAs are performed. If the DSCA= operand is specified for 3270P, it is ignored, except for the bit setting for "sound device alarm." If this bit is specified on the DSCA/SCA option, it is sent to the device.

The value specified here must be a decimal number not exceeding 65535 or X'hhhh'. If the number is specified, the number is internally converted to X'hhhh'.

If byte 1 bit 5 is set to B'1' (unprotect screen option) for a 3275 display, and both input and output occur simultaneously (contention), the device is disconnected. For non-3275 devices, the SCA option is ignored. If byte 1 bit 5 is set to B'0', the application program can request autopaged output by setting the SCA value to B'1'. This request is honored only if present in the first segment of the first LPAGE of the output message.

If a nonzero value is specified for byte 0, or for bit 6 or 7 in byte 1, MFS overrides the specified value with zero.

### *features : MFSFeatureType*

*FEAT attribute*

Specifies features for this device or program group. Possible features are:

- IGNORE specifies that device features are to be ignored for this device.
- 120|126|132 specifies line length for 3284, and 3286 device types (TYPE=3270P).
- CARD specifies that the device has a 3270 operator identification card reader. NOCD specifies the absence of the CARD feature.
- DEKYBD specifies data entry keyboard feature. This feature implies PFK feature; therefore, PFK is invalid if DEKYBD is specified. NOPFK implies the absence of PFK and DEKYBD features.
- PFK specifies that the device has program function keys. NOPFK specifies the absence of the PFK and DEKYBD features.
- PEN specifies the selector light pen detect feature. NOPEN specifies the absence of the PEN feature.
- 1|2|3|4|5|6|7|8|9|10 specifies customer-defined features for the 3270P device type.

For 3270P devices, FEAT= allows grouping of devices with special device characteristics. For example, FEAT=1 could group devices with a maximum of 80 print positions and no VFC, and FEAT=2 could group devices with 132 print positions and the VFC feature. FEAT=IGNORE should be specified to group together devices with a minimum set of device capabilities. When WIDTH= is specified, FEAT=(1...10) must also be specified. If FEAT=(1...10) is specified but WIDTH= is not specified, WIDTH= defaults to 120.

When IGNORE is specified, no other values should be coded in the FEAT= operand. When FEAT=IGNORE is not specified in the TERMINAL macro during system definition, the MSG statement must specify IGNORE in the SOR= operand for the device format with the IGNORE specification. Unless FEAT=IGNORE is used, FEAT= must specify exactly what was specified in the TERMINAL macro during IMS system definition. If it does not, the DFS057 error message is issued. When FEAT=IGNORE or 1-10 is specified for 3270 devices, the operands PEN=, CARD=, and PFK= can still be specified. When TYPE=3270P and FEAT=IGNORE, MFS allows a line width of 120 characters.

CARD, PFK, DEKYBD, and PEN feature values are valid only for 3270 displays. If the FEAT= operand is omitted, the default features are CARD, PFK, and PEN for 3270 displays; the default line width is 120 for TYPE=3270P.

1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are valid values only for 3270, 3270P and 3270-An. For 3270 displays, the FEAT= specifications of 1 to 5 can be used to group devices with specific features or hardware data stream dependencies.

Restriction: This keyword is optional and cannot be used with any other feature specification for 3270 displays.

Feature operand values can be specified in any order, and only those values desired need be specified. The underlined values do not have to be specified because they are defaults. Only one value in each vertical list can be specified.

### *page : MFSPageType*

*PAGE attribute*

Specifies output parameters as follows:

- number: For printer devices, number defines the number of print lines on a printed page; for card devices, number defines the number of cards to be punched per DPAGE or physical page (if pp parameter is used in the DFLD statements). This value is used for validity checking. The number specified must be greater than or equal to 1 and less than 256. The default is 55.
- DEFN specifies that lines/cards are to be printed/punched as defined by DFLD statements (no lines/cards are to be removed or added to the output page).
- SPACE specifies that each output page contains the exact number of lines/cards specified in the number parameter.
- FLOAT specifies that lines/cards with no data (all blank or NULL) after formatting are to be deleted.
- For 3270P devices, some lines having no data (that is, all blank or null) must not be deleted under the following circumstances:
  - The line contains one or more set line density (SLDx=) specifications.
  - A field specified as having extended attributes spans more than one line.

### *pen : 0..1 MFSDeviceField*

*PEN attribute*

Defines an input field name to contain literal data when an immediate light pen detection of a field with a space or null designator character occurs. The literal data is defined on the DFLD statement with the PEN= operand. (See PEN= operand on the DFLD statement.) This name can be referred to by an MFLD statement and must not be used as the label of a DFLD statement within this DEV definition. The PEN= operand is valid only for 3270 displays. If FEAT=NOPEN is specified, it is changed to PEN.

If an immediate detect occurs on a field defined with a space or null designator character, and either another field has been selected or modified or has the MOD attribute, or the PEN= operand is not defined for the DFLD, a question mark (?) is inserted in the PEN= field name.

If no immediate detection occurs or the immediate detect occurs on a field defined with an ampersand (&) designator character, the PEN= operand is padded with the fill specified in the MFLD statement.

### *pfk : MFSFunctionKeyType*

*PFK attribute*

Defines an input field name to contain program function key literal or control function data (first subparameter) and, in positional or keyword format, either the literal data to be placed in the specified field, or the control function to be performed when the corresponding function key is entered (remaining subparameters).

The name of the first subparameter (the input field name that will contain the program function key literal or control function data) can be referred to by an MFLD statement and must not be used as the label of a DFLD statement within this DEV definition. The remaining subparameters can be specified in positional or keyword format. If the subparameters are in keyword format, the integer specified must be from 1 to 36, inclusive, and not duplicated. Only one PFK= operand format (positional or keyword) can be specified on a DEV statement. This operand is valid only for 3270 displays. At the time the actual format blocks are created, each literal is padded on the right with blanks to the length of the largest literal in the list. The maximum literal length is 256 bytes.

If the device supports the IMS copy function, then PFK12 invokes the copy function and the definition of PFK12 in the DEV statement is ignored; otherwise, the definition of PFK12 is honored.

If FEAT=NOPFK is specified, it is changed to PFK. The maximum number of user-defined PFKs is 36.

Control functions that can be specified are:

- NEXTPP--PAGE ADVANCE specifies a request for the next physical page in the current output message. If no output message is in progress, no explicit response is made.
- NEXTMSG--MESSAGE ADVANCE specifies a request to dequeue the output message in progress (if any) and to send the next output message in the queue (if any).
- NEXTMSGP--MESSAGE ADVANCE PROTECT specifies a request to dequeue the output message in progress (if any), and send the next output message or return an information message indicating that no next message exists.
- NEXTLP--NEXT LOGICAL PAGE specifies a request for the next logical page of the current message.

- ENDMPPI--END MULTIPLE PAGE INPUT specifies the end of a multiple physical page input message.

### *substitution : String*

*SUB attribute*

Specifies the character used by MFS to replace any X'3F' characters in the input data stream. No translation occurs if this parameter is specified as X'3F' or this parameter is not specified, or the input received bypasses MFS editing. The specified SUB character should not appear elsewhere in the data stream; therefore, it should be non-graphic.

- X'hh' character whose hexadecimal representation is 'hh' replaces all X'3F' in the input data stream.
- C'c' character 'c' replaces all X'3F' in the input data stream.

### *systemMessage : 0..* MFSDeviceField*

*SYSMSG attribute*

Specifies the label of the DFLD statements that define the device field in which IMS system messages are to be displayed. This operand is valid only if a 3270 display is specified. A DFLD with this label should be defined for each physical page within each DPAGE defined within this DEV definition. DFLDs for SYSMSG should be at least LTH=79 to prevent message truncation. The referenced DFLD can also be referenced by an MFLD statement.

### *type : String*

*TYPE attribute*

Specifies the device type and model number of a device using this format description. The 3284-3 printer attached to a 3275 is supported only as TYPE=3270P. The model number specified when defining a format for a 3284-3 is the model number of the associated 3275.

TYPE=3270-An specifies a symbolic name for 3270 and SLU 2 displays with the screen size defined during IMS system definition, feature numbers n=1-15. This specification causes the MFS Language utility to read the MFS device characteristics table (DFSUDT0x) to extract the screen size.

### *width : int*

*WIDTH attribute*

Specifies the maximum line width for this DEV type as one of:

- Number of print positions per line of input or output data
- Number of punch positions per card of input or output data
- Card width for card reader input data

The default is 120 for 3270P output. Line width is specified relative to column 1, regardless of whether a left margin value is specified in the HTAB= keyword. The width specified must be greater than or equal to 1.

For 3270P devices, if WIDTH is specified, then FEAT=(1...10) must also be specified. If FEAT=(1...10) is specified, and WIDTH= is not specified, WIDTH= defaults to 120.

## *MFSIfCondition*

This class encapsulates the MFS "IF" statement.

The IF statement defines an entry in the table named by the previous TABLE statement. Each IF statement defines a conditional operation and an associated control or branching function to be performed if the condition is true. All attributes are supported.

### ***condition : MFSConditionType***

*COND attribute*

condition has the following format:

IF (DATA | LENGTH) (=,<,>, ,×,× )

(literal | data-length) function:

- DATA specifies that the conditional operation is to be performed against the data received from the device for the field.
- LENGTH specifies that the conditional operation is testing the number of characters entered for the field. The size limit for this field is the same as for DFLDs (see "DFLD Statement" in topic 2.5.1.5.8).
- =,<,>, ,×,× specify the conditional relationship that must be true to invoke the specified control function.
- 'literal' is a literal string to which input data is to be compared. The compare is done before the input is translated to upper case. If 'literal' is specified, DATA must be specified in the first operand. If the input data length is not equal to the literal string length, the compare is performed with the smaller length, unless the conditional relationship is and the data length is zero, in which case the control function is performed. If the input is in lowercase, the ALPHA statement should be used and the literal coded in lowercase.
- data-length specifies an integer value to which the number of characters of input data for the field is compared.
- NOFUNC specifies that conditional function testing is to be terminated.
- NEXTPP--PAGE ADVANCE specifies a request for the next physical page in the current output message. If no output message is in progress, no explicit response is made.

- NEXTMSG--MESSAGE ADVANCE specifies a request to dequeue the output message in progress (if any) and to send the next output message in the queue (if any).
- NEXTMSGP--MESSAGE ADVANCE PROTECT specifies a request to dequeue the output message in progress (if any), and either send the next output message or return an information message indicating that no next message exists.
- NEXTLP--NEXT LOGICAL PAGE specifies a request for the next logical page of the current message.
- PAGEREQ--LOGICAL PAGE REQUEST specifies that the second through last characters of input data are to be considered as a logical page request.
- ENDMPPI--END MULTIPLE PAGE INPUT specifies the end of multiple physical page input (this input is the last for the message being created).

#### *action : String*

*COND attribute*

Contains the 'function' described above.

### *MFSLogicalPage*

This class encapsulates the MFS "LPAGE" statement.

The optional LPAGE statement defines a group of segments comprising a logical page. It is implied if not present. All attributes are supported.

#### *condition : MFSConditionType*

*COND attribute*

Describes a conditional test that, if successful, specifies that the segment and field definitions following this LPAGE are to be used for output editing of this logical page. The specified portion of the first segment of a logical page is examined to determine if it is greater than (>), less than (<), greater than or equal to (×), less than or equal to (×), equal to (=), or not equal to (ne) the specified literal value to determine if this LPAGE is to be used for editing. COND= is not required for the last LPAGE statement in the MSG definition.

The area examined can be defined by a field name (mfldname), an offset in a field (mfldname(pp) where pp is the offset in the named field), or an offset in the segment (segoffset). If the mfldname(pp) form is used, pp must be greater than or equal to 1. The length of the compare is the length of the specified literal. If OPT=3 is specified on the previous MSG statement, the area to be examined must be within one field as defined on an MFLD statement.

If segoffset is used, it is relative to zero, and the specification of that offset must allow for LLZZ of the segment (that is, the first data byte is at offset 4).

If pp is used, the offset is relative to 1 with respect to the named field (that is, the first byte of data in the field is at offset 1, not zero).

If the mfldname specified is defined with ATTR=YES, the pp offset must be used. The minimum offset specified must be 3. That is, the first byte of data in the field is at offset 3, following the two bytes of attributes.

If ATTR=nn is specified, the minimum offset must be one plus twice nn. Thus, if ATTR=2 is specified, pp must be at least 5, and, if ATTR=(YES,2) is specified, pp must be at least 7.

If the conditional tests for all LPAGEs fail, the last LPAGE in this MSG definition is used for editing.

If LPAGE selection is to be specified using the command data field, that is, /FORMAT modname...(data), the MFLD specified in the LPAGE COND=mfldname parameter should be within the first 8 bytes of the associated LPAGEs of the MOD.

#### *prompt : 0..1 MFSDeviceField*

*PROMPT attribute*

Specifies the name of the DFLD into which MFS should insert the specified literal when formatting the last logical page of an output message. If FILL=NULL is specified once the prompt literal is displayed, it can remain on the screen if your response does not cause the screen to be reformatted.

### *MFSMessageDescriptor*

This class encapsulates the MFS "MSG" statement.

The MSG statement initiates and names a message input or output definition. All attributes are supported.

#### *fill : String*

*FILL attribute*

Specifies a fill character for output device fields. This operand is valid only if TYPE=OUTPUT. The default is C' '. The fill specification is ignored unless FILL=NONE is specified on the DPAGE statement in the FMT definition. For 3270 output when EGCS fields are present, only FILL=PT or FILL=NULL should be specified. A FILL=PT erases an output field (either a 1- or 2-byte field) only when data is sent to the field, and thus does not erase the DFLD if the application program message omits the MFLD.

- Character 'c' is used to fill device fields. For 3270 display devices, any specification with a value less than X'3F' is changed to X'00' for control characters or to X'40' for other non-graphic characters. For all other devices, any FILL=C'c' specification with a value less than X'3F' is ignored and defaulted to X'3F' (which is equivalent to a specification of FILL=NULL).
- NULL specifies that fields are not to be filled.

- PT is identical to NULL except for 3270 display. For 3270 display, PT specifies that output fields that do not fill the device field (DFLD) are followed by a program tab character to erase data previously in the field.

#### *ignoreSource : Boolean*

*SOR attribute*

Specifies the source name of the FMT statement, which, with the DEV statement, defines the terminal or remote program data fields processed by this message descriptor. Specifying IGNORE for TYPE=OUTPUT causes MFS to use data fields specified for the device whose FEAT= operand specifies IGNORE in the device format definition. For TYPE=INPUT, IGNORE should be specified only if the corresponding message output descriptor specified IGNORE. If you use SOR=IGNORE, you must specify IGNORE on both the message input descriptor and the message output descriptor.

#### *option : int*

*OPT attribute*

Specifies the message formatting option used by MFS to edit messages. The default is 1.

#### *paging : Boolean*

*PAGE attribute*

Specifies whether (YES) or not (NO) operator logical paging (forward and backward paging) is to be provided for messages edited using this control block. This operand is valid only if TYPE=OUTPUT. The default is NO, which means that only forward paging of physical pages is provided.

#### *type : MFSDescriptorType*

*TYPE attribute*

Defines this definition as a message INPUT or OUTPUT control block. The default is INPUT.

### *MFSMessageField*

This class encapsulates the MFS "MFLD" statement.

The MFLD statement defines a message field as it will be presented to an application program as part of a message output segment. At least one MFLD statement must be specified for each MSG definition. All attributes are supported.

#### *attributes : Boolean*

*ATTR attribute*

Specifies whether the application program can (YES) modify or cannot (NO) modify the 3270 attributes and the extended attributes (nn).

If YES, 2 bytes must be reserved for the 3270 attribute data to be filled in by the application program on output and to be initialized to blanks on input. These 2 bytes must be included in the LTH=specification.

The value supplied for nn is the number of extended attributes that can be dynamically modified. The value of nn can be a number from 1 to 6. An invalid specification will default to 1. Two additional bytes per attribute must be reserved for the extended attribute data to be filled in by the application program on output and to be initialized to blanks on input. These attribute bytes must be included in the MFLD LTH= specification.

Example: Shown below are valid specifications for ATTR= and the number of bytes that must be reserved for each different specification:

| **Specifications** | **Number of Bytes** |
|---|---|
| MFLD ,ATTR=(YES,nn) | 2 + (2 Þ nn) |
| MFLD ,ATTR=(NO,nn) | 2 Þ nn |
| MFLD ,ATTR=(nn) | 2 Þ nn |
| MFLD ,ATTR=YES | 2 |
| MFLD ,ATTR=NO | 0 |

ATTR=YES and nn are invalid if a literal value has been specified through the positional parameter in an output message.

The attributes in a field sent to another IMS ISC subsystem are treated as input data by MFS regardless of any ATTR= specifications in the format of the receiving subsystem. For example, a message field (MFLD) defined as ATTR=(YES,1),LTH=5 would contain the following:

    00A0C2F1C8C5D3D3D6

If the MFLD in the receiving subsystem is defined as LTH=9 and without ATTR=, the application program receives:

    00A0C2F1C8C5D3D3D6

If the MFLD in the receiving subsystem is defined as LTH=13 and ATTR=(YES,1), the application program receives:

    4040404000A0C2F1C8C5D3D3D6

If the MFLD in the receiving subsystem is defined as LTH=5 and ATTR=(YES,1), the application program receives:

    4040404000A0C2F1C8

The input SEG statement should be specified as GRAPHIC=NO to prevent translation of the attribute data to uppercase.

#### *exit : MFSExitType*

*EXIT attribute*

Describes the field edit exit routine interface for this message field. The exit routine number is specified in exitnum, and exitvect is a value to be passed to the exit routine when it is invoked for this field. The value of exitnum can range from 0 to 127. The value of exitvect can range from 0 to 255. The address of the field as it exists after MFS editing, (but before NULL compression for option 1 and 2), is passed to the edit exit routine, along with the vector defined for the field. (If NOFLDEXIT is specified for a DPM device, the exit routine will not be invoked.) The exit routine can return a code with a value from 0 to 255. MFS maintains the highest such code returned for each segment for use by the segment edit routine. EXIT= is invalid if 'literal' is specified on the same MFLD statement.

#### *extendedAttributes : Boolean*

ATTR attribute.

See attributes documentation above.

#### *fill : String*

*FILL attribute*

Specifies a character to be used to pad this field when the length of the data received from the device is less than the length of this field. This character is also used to pad when no data is received for this field (except when MSG statement specifies option 3.) This operand is only valid if TYPE=INPUT. The default is X'40'.

- X'hh' - Character whose hexadecimal representation is hh is used to fill fields. FILL=X'3F' is the same as FILL=NULL.
- C'c' - Character c is used to fill fields.
- NULL causes compression of the message segment to the left by the amount of missing data in the field.

#### *justify : MFSJustifyType*

*JUST attribute*

Specifies that the data field is to be left-justified (L) or right-justified (R) and right- or left- truncated as required, depending upon the amount of data expected or presented by the device format control block. The default is L.

#### *length : MFSLengthType*

*LTH attribute*

Length can be omitted if a literal is specified in the positional operand (TYPE=INPUT), in which case, length specified for literal is used. If LTH= is specified for a literal field, the specified literal is either truncated or padded with blanks to the specified length. If the MFLD statement appears between a DO and an ENDDO statement, a length value is printed on the generated MFLD statement, regardless of whether LTH= is specified in the MFLD source statement.

### *value : String*

Corresponds to the 'literal' field in the following description.

The device field name is specified via the 'deviceFields' relationship.

Specifies the device field name (defined via the DEV or DFLD statement) from which input data is extracted or into which output data is placed. If this parameter is omitted when defining a message output control block, the data supplied by the application program is not displayed on the output device. If the repetitive generation function of MFS is used (DO and ENDDO statements), dfldname should be restricted to 6 characters maximum length. When each repetition of the statement is generated, a 2-character sequence number (01 to 99) is appended to dfldname. If the dfldname specified here is greater than 6 bytes and repetitive generation is used, dfldname is truncated at 6 characters and a 2-character sequence number is appended to form an 8-character name. No error message is provided if this occurs. This parameter can be specified in one of the following formats:

- dfldname identifies the device field name from which input data is extracted or into which output data is placed.
- 'literal' can be specified if a literal value is to be inserted in an input message.

    (dfldname,'literal')

  If TYPE=OUTPUT, this describes the literal data to be placed in the named DFLD. When this form is specified, space for the literal must not be allocated in the output message segment supplied by the

    application program.

  If TYPE=INPUT, this describes the literal data to be placed in the message field when no data for this field is received from the device. If this dfldname is used in the PFK parameter of a DEV statement, this literal is always replaced by the PF key literal or control function. However, when this dfldname is specified in the PFK parameter, but the PF key is not used, the literal specified in the MFLD statement is moved into the message field. When physical paging is used, the literal is inserted in the field but is not processed until after the last physical page of the logical page has been displayed.

  In both cases, if the LTH= operand is specified, the length of the literal is truncated or padded as necessary to the length of the LTH= specification. If the length of the specified literal is less than the defined field length, the literal is padded with blanks if TYPE=OUTPUT and with the specified fill character (FILL=) if TYPE=INPUT. If no fill character is specified for input, the literal is padded with blanks (the default). The length of the literal value cannot exceed 256 bytes.

(dfldname,system-literal) specifies a name from a list of system literals. A system literal functions like a normal literal except that the literal value is created during formatting prior to transmission to the device. The LTH=, ATTR=, and JUST= operands cannot be specified. When this form is specified, space for the literal must not be allocated in the output message segment supplied by the application program.

(,SCA) defines this output field as the system control area, which is not displayed on the output device. There can be only one such field in a logical page (LPAGE) and it must be in the first message segment of that page. If no logical pages are defined, only one SCA field can be defined and it must be in the first segment of the output message. This specification is valid only if TYPE=OUTPUT was specified on the previous MSG statement.

### *MFSPassword*

This class encapsulates the MFS "PASSWORD" statement.

The PASSWORD statement identifies one or more fields to be used as an IMS password. When used, the PASSWORD statement and its associated MFLDs must precede the first SEG statement in an input LPAGE or MSG definition. Up to 8 MFLD statements can be specified after the PASSWORD statement but the total password length must not exceed 8 characters. The fill character must be X'40'. For option 1 and 2 messages, the first 8 characters of data after editing are used for the IMS password. For option 3 messages, the data content of the first field after editing is used for the IMS password.

A password for 3270 input can also be defined in a DFLD statement. If both password methods are used, the password specified in the MSG definition is used. All attributes are supported.

### *MFSSegment*

This class encapsulates the MFS "SEG" statement.

The SEG statement delineates message segments and is required only if multisegment message processing is required by the application program. Output message segments cannot exceed your specified queue buffer length. Only one segment should be defined for TYPE=INPUT MSGs when the input message destination is defined as a single segment command or transaction. If more than one segment is defined, and the definition is used to input a single segment command or transaction, care must be used to ensure that your input produces only one segment after editing. It is implied if not present. All attributes are supported.

#### *exit : MFSExitType*

*EXIT attribute*

Describes the segment edit exit routine interface for this message segment. exitnum is the exit routine number and exitvect is a value to be passed to the exit routine when it is invoked for this segment. exitnum can range from 0 to 127. exitvect can range from 0 to 255. The SEG exit is invoked when processing completes for the input segment.

### *graphic : Boolean*

*GRAPHIC attribute*

Specifies for MSG TYPE=INPUT whether (YES) or not (NO) IMS should perform upper case translation on this segment if the destination definition requests it (see the EDIT= parameter of the TRANSACT or NAME macro). The default is YES. If input segment data is in non-graphic format (packed decimal, EGCS, binary, and so forth), GRAPHIC=NO should be specified. When GRAPHIC=NO is specified, FILL=NULL is invalid for MFLDs within this segment.

The list below shows the translation that occurs when GRAPHIC=YES is specified and the input message destination is defined as requesting upper case translation.

| **Before Translation** | **After Translation** |
|---|---|
| a through z | A through Z |
| X'81' through X'89 | 'X'C1' through X'C9' |
| X'91' through X'99 | 'X'D1' through X'D9' |
| X'A2' through X'A9 | 'X'E2' through X'E9' |

If FILL=NULL is specified for any MFLD in a segment defined as GRAPHIC=YES, the hexadecimal character X'3F' is compressed out of the segment. If GRAPHIC=NO and FILL=NULL are specified in the SEG statement, any X'3F' in the non-graphic data stream is compressed out of the segment and undesirable results might be produced. Non-graphic data should be sent on output as fixed length output fields and the use of FILL=NULL is not recommended in this case.

For MSG TYPE=OUTPUT, the GRAPHIC= keyword applies only for DPM. It specifies whether (YES) or not (NO) non-graphic control characters (X'00' to X'3F') in the data from the IMS application program are to be replaced by blanks. The default value is YES. If NO is specified, MFS allows any bit string received from an IMS application program to flow unmodified through MFS to the remote program.

Restriction: When GRAPHIC=NO is specified, IMS application programs using Options 1 and 2 cannot omit segments in the middle of an LPAGE, or truncate or omit fields in the segment using the null character (X'3F').

### *MFSTable*

This class encapsulates the MFS "TABLE" statement.

The TABLE statement initiates and names an operator control table that can be referred to by the OPCTL keyword of the DFLD statement. The TABLE statement, and the IF and TABLEEND statements that follow, must be outside of an MSG or FMT definition. All attributes are supported.

## A.3 CICS BMS Metamodel

CICS applications are able to use a logical abstraction of a terminal datastream using CICS Basic Mapping Support (BMS) function. Its highest use is with the IBM3270 family of alphanumeric displays and associated printers but does support other devices and MQ queues. The programmer creates an input file containing the variable data from the application to be displayed on output or formatted on input plus the constant 'boilerplate' that should appear on the screen. Each field can have attributes added to it, for example, color, protection so that it cannot be overwritten by the operator and various productivity options such as cursor positioning and auto-skipping to the next input field. These fields are aggregated together into a MAP. MAPs may also be aggregated into MAPSETs.

The input file is pre-processed to provide an application structure that will be included with the CICS application program giving the programmer fields in which to place the variable data, and secondly produces a file that contains all the constant data and the attributes of each field. A simple view of this is that the BMS input file has the same attributes as an HTTP data, formatting commands are mixed with the data, the output of the BMS processor is almost a parallel with XML and XSL, the data structure holding the data items and the file holding all the style information. Unfortunately there are two pieces of state data held in the BMS 'style' sheet, namely the initial cursor position and an attribute declaration that will force the terminal to return the data on the screen whether or not the operator has changed it. When an EXEC CICS SEND MAP is performed, BMS will interpret the map file and merge in the data from the application structure and any overridden attributes, and build the device dependent data stream required for the terminal. Conversely on an EXEC CICS RECEIVE MAP the inbound datastream is mapped into the application structure with whatever filling or conversion that is required.

The CICS BMS metamodel captures the meta data associated with screen formatting for CICS applications. BMSField identifies the signature of a CICS BMS message, which can include inputs, outputs, and return types. BMSField associates with TDLangElement, which provides the linkage to the language specific and physical representations of the data that a BMSField represents. The following figures illustrate the classes that constitute the CICS BMS metamodel and show how the classes relate to each other. Following the diagrams is a brief explanation of what each class represents.

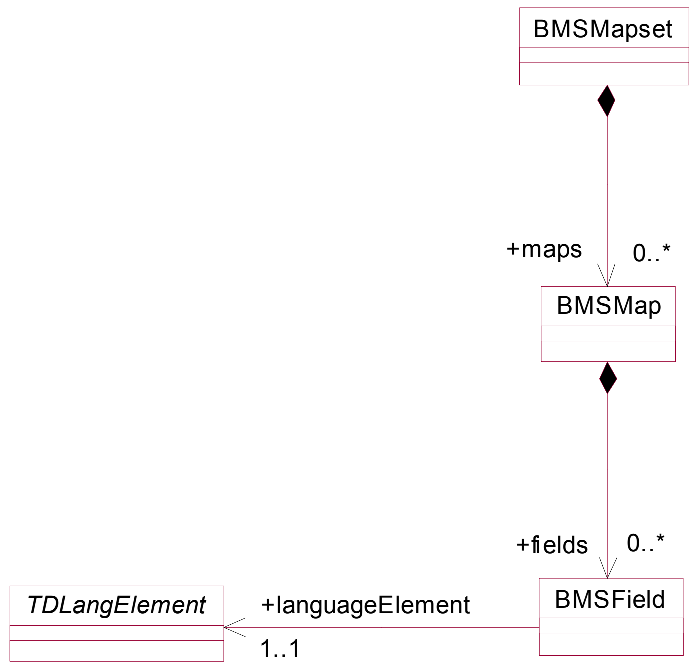



*Figure A-10* CICS BMS Relationship View

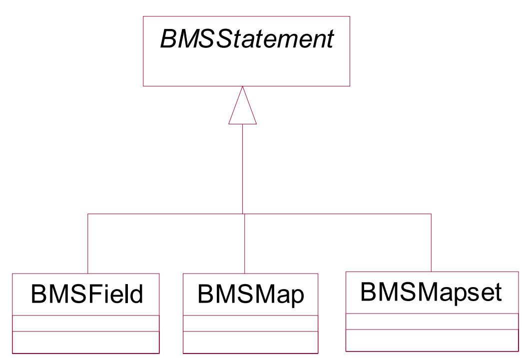



*Figure A-11* CICS BMS Inheritance View

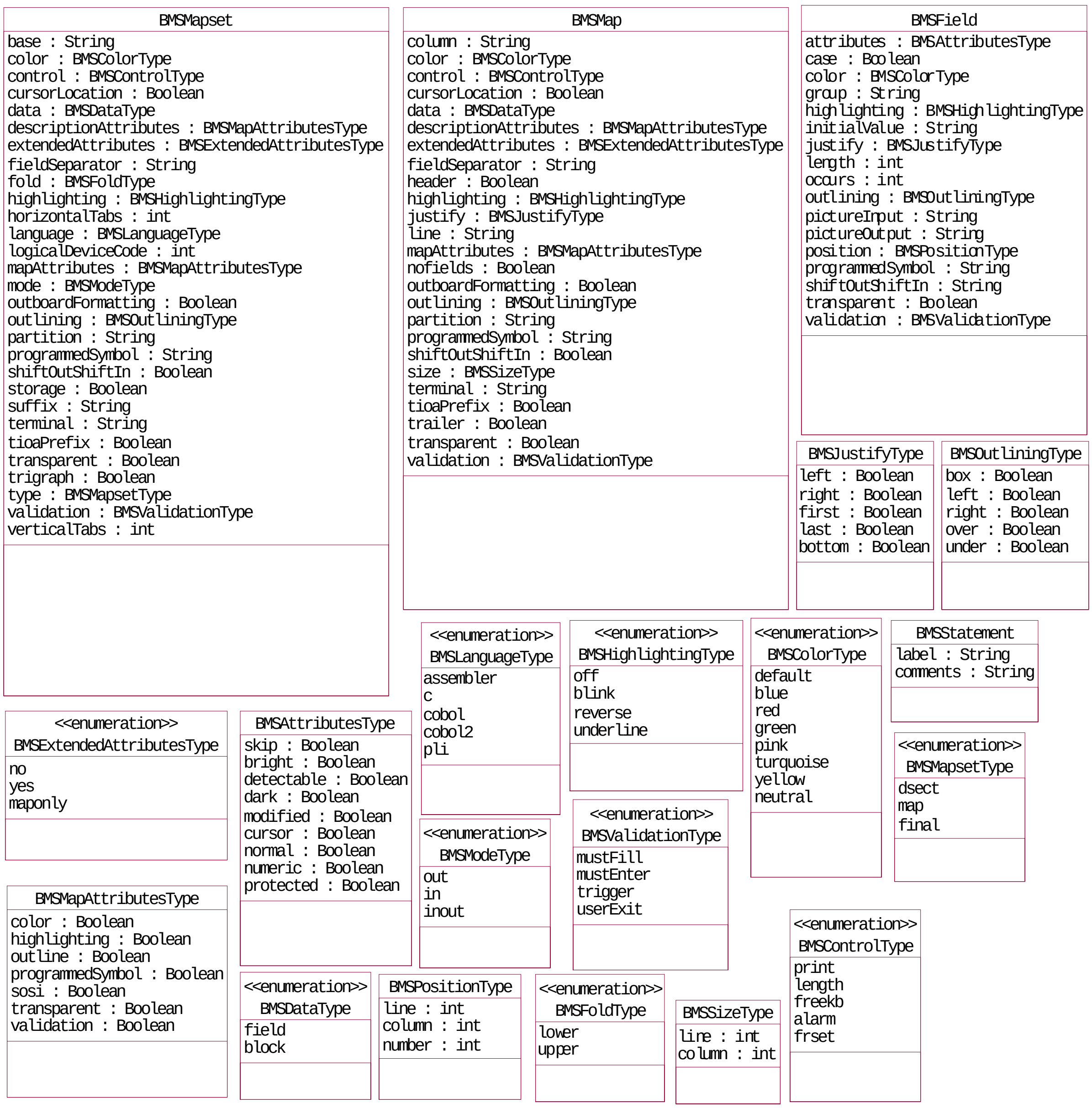



*Figure A-12* CICS BMS Attributes

## A.3.1 CICS BMS Metamodel Descriptions

### BMSAttributesType

BMSAttributesType is the ATTRB statement. This operand applies only to 3270 data stream devices; it is ignored for other devices, except that ATTRB=DRK is honored for the SCS Printer Logical Unit. It is also ignored (except for ATTRB=DRK) if the NLEOM option is specified on the SEND MAP command for transmission to a 3270 printer. In particular, ATTRB=DRK should not be used as a method of protecting secure data on output on non-3270, non-SCS printer terminals.

If ATTRB is specified within a group of fields, it must be specified in the first field entry. A group of fields appears as one field to the 3270. Therefore, the ATTRB specification refers to all of the fields in a group as one field rather than as individual fields. It specifies device-dependent characteristics and attributes, such as the capability of a field to receive data, or the intensity to be used when the field is output. It could however, be used for making an input field non-display for secure entry of a password from a screen.

For input map fields, DET and NUM are the only valid options; all others are ignored.

ASKIP is the default and specifies that data cannot be keyed into the field and causes the cursor to skip over the field.

BRT specifies that a high-intensity display of the field is required. Because of the 3270 attribute character bit assignments, a field specified as BRT is also potentially light pen detectable. However, for the field to be recognized as detectable by BMS, DET must also be specified.

- DET specifies that the field is potentially detectable. The first character of a 3270 detectable field must be one of the following:

  ? > & blank

  If ? or >, the field is a selection field; if & or blank, the field is an attention field. (See “An Introduction to the IBM 3270 Information Display System” for further details about detectable fields.)

  A field for which BRT is specified is potentially detectable to the 3270, because of the 3270 attribute character bit assignments, but is not recognized as such by BMS unless DET is also specified.

  DET and DRK are mutually exclusive. If DET is specified for a field on a map with MODE=IN, only one data byte is reserved for each input field. This byte is set to X'00', and remains unchanged if the field is not selected. If the field is selected, the byte is set to X'FF'.

  No other data is supplied, even if the field is a selection field and the ENTER key has been pressed.

  If the data in a detectable field is required, all of the following conditions must be fulfilled:

1. The field must begin with one of the following

   characters:

   ? > & blank

   and DET must be specified in the output map.

2. The ENTER key (or some other attention key) must be pressed after the field has been selected, although the ENTER key is not required for detectable fields beginning with & or a blank.

3. DET must not be specified for the field in the input map. DET must, however, be specified in the output map. For more information about BMS support of the light pen, see the CICS Application Programming Guide.

- DRK specifies that the field is non-print/non-display. DRK cannot be specified if DET is specified.

  FSET specifies that the modified data tag (MDT) for this field should be set when the field is sent to a terminal. Specification of FSET causes the 3270 to treat the field as though it has been modified. On a subsequent read from the terminal, this field is read, whether or not it has been modified. The MDT remains set until the field is rewritten without ATTRB=FSET, or until an output mapping request causes the MDT to be reset.

  Either of two sets of defaults may apply when a field to be displayed on a 3270 is being defined but not all parameters are specified. If no ATTRB parameters are specified, ASKIP and NORM are assumed. If any parameter is specified, UNPROT and NORM are assumed for that field unless overridden by a specified parameter.

- IC specifies that the cursor is to be placed in the first position of the field. The IC attribute for the last field for which it is specified in a map is the one that takes effect. If not specified for any fields in a map, the default location is zero. Specifying IC with ASKIP or PROT causes the cursor to be placed in an un-keyable field.

  This option can be overridden by the CURSOR option of the SEND MAP command that causes the write operation.

- NORM specifies that the field intensity is to be normal.

- NUM ensures that the data entry keyboard is set to numeric shift for this field unless the operator presses the alpha shift key, and prevents entry of nonnumeric data if the Keyboard Numeric Lock feature is installed.

- PROT specifies that data cannot be keyed into the field. If data is to be copied from one device to another attached to the same 3270 control unit, the first position (address 0) in the buffer of the device to be copied from must not contain an attribute byte for a protected field. Therefore, when preparing maps for 3270s, ensure that the first map of any page does not contain a protected field starting at position 0.

- UNPROT specifies that data can be keyed into the field.

## *BMSColorType*

BMSColorType indicates the individual color, or the default color for the mapset (where applicable). The valid colors are blue, red, pink, green, turquoise, yellow, and neutral. The COLOR operand is ignored unless the terminal supports color.

## *BMSControlType*

BMSControlType is the CTRL statement. It defines characteristics of IBM 3270 terminals. Use of any of the control options in the SEND MAP command overrides all control options in the DFHMDI macro, which in turn overrides all control options in the DFHMSD macro.

If CTRL is used with cumulative BMS paging (that is, the ACCUM option is used on the BMS SEND MAP commands), it must be specified on the last (or only) map of a page, unless it is overridden by the ALARM, FREEKB and so on, options on the SEND MAP or accumulated SEND CONTROL command.

PRINT must be specified if the printer is to be started; if omitted, the data is sent to the printer buffer but is not printed. This operand is ignored if the mapset is used with 3270 displays without the Printer Adapter feature.

LENGTH indicates the line length on the printer; length can be specified as L40, L64, L80, or HONEOM. L40, L64, and L80 force a new line after 40, 64, or 80 characters, respectively. HONEOM causes the default printer line length to be used. If this option is omitted, BMS sets the line length from the terminal definition page size.

FREEKB causes the keyboard to be unlocked after the map is written. If FREEKB is not specified, the keyboard remains locked; data entry from the keyboard is inhibited until this status is changed.

ALARM activates the 3270 audible alarm if available.

FRSETspecifies that the modified data tags (MDTs) of all fields currently in the 3270 buffer are to be reset to an unmodified condition (that is, field reset) before map data is written to the buffer. This allows the DFHMDF macro with the ATTRB operand to control the final status of any fields written or rewritten in response to a BMS command.

Note: CTRL cannot be specified in the DFHMDI and DFHMSD macros in the same mapset.

## *BMSDataType*

BMSDataType can be either "field" or "block."

## *BMSExtendedAttributesType*

BMSExtendedAttributesType can be "no," "yes," or "maponly."

### *BMSField*

BMSField is implemented by the DFHMDF macro. BMSField has the following attributes:

- GRPNAME is the name used to generate symbolic storage definitions and to combine specific fields under one group name. The same group name must be specified for each field that is to belong to the group. The length of the name is up to 30 characters though you should refer to the compiler manual to make sure that there are no other restrictions on the length. If this operand is specified, the OCCURS operand cannot be specified.

  The fields in a group must follow on; there can be gaps between them, but not other fields from outside the group. A field name must be specified for every field that belongs to the group, and the POS operand must also be specified to ensure that the fields follow each other. All the DFHMDF macros defining the fields of a group must be placed together, and in the correct order (ascending numeric order of the POS value).

  For example, the first 20 columns of the first six lines of a map can be defined as a group of six fields, as long as the remaining columns on the first five lines are not defined as fields.

- attributes is the ATTRB operand specified on the first field of the group applies to all of the fields within the group.

- length is the LENGTH operand. It specifies the length (1-256 bytes) of the field or group of fields. This length should be the maximum length required for application program data to be entered into the field; it should not include the one-byte attribute indicator appended to the field by CICS for use in subsequent processing. The length of each individual subfield within a group must not exceed 256 bytes. LENGTH can be omitted if PICIN or PICOUT is specified, but is required otherwise. You can specify a length of zero only if you omit the label (field name) from the DFHMDF macro. That is, the field is not part of the application data structure and the application program cannot modify the attributes of the field. You can use a field with zero length to delimit an input field on a map.

  The map dimensions specified in the SIZE operand of the DFHMDI macro defining a map can be smaller than the actual page size or screen size defined for the terminal.

  If the LENGTH specification in a DFHMDF macro causes the map-defined boundary on the same line to be exceeded, the field on the output screen is continued by wrapping.

- occurs is the OCCURS operand. It specifies that the indicated number of entries for the field are to be generated in a map, and that the map definition is to be generated in such a way that the fields are addressable as entries in a matrix or an array. This permits several data fields to be addressed by the same name (subscripted) without generating a unique name for each field.

OCCURS and GRPNAME are mutually exclusive; that is, OCCURS cannot be used when fields have been defined under a group name. If this operand is omitted, a value of OCCURS=1 is assumed.

- pictureInput is the PICIN operand (COBOL and PL/I only). It specifies a picture to be applied to an input field in an IN or INOUT map; this picture serves as an editing specification that is passed to the application program, thus permitting the user to exploit the editing capabilities of COBOL or PL/I. BMS checks that the specified characters are valid picture specifications for the language of the map.

  However, the validity of the input data is not checked by BMS or the high-level language when the map is used, so any desired checking must be performed by the application program. The length of the data associated with "value" should be the same as that specified in the LENGTH operand if LENGTH is specified. If both PICIN and PICOUT are used, an error message is produced if their calculated lengths do not agree; the shorter of the two lengths is used. If PICIN or PICOUT is not coded for the field definition, a character definition of the field is automatically generated regardless of other operands that are coded, such as ATTRB=NUM.

  Note: The valid picture values for COBOL input maps are:

  A P S V X 9 / and (

  The valid picture values for PL/I input maps are:

  A B E F G H I K M P R S T V X Y and Z

  1 2 3 6 7 8 9 / + - , . * $ and (

  For PL/I, a currency symbol can be used as a picture character. The symbol can be any sequence of characters enclosed in < and >, for example <DM>.

  Refer to the appropriate language reference manual for the correct syntax of the PICTURE attribute.

- pictureOutput is the PICOUT operand (COBOL and PL/I only). It is similar to PICIN, except that a picture to be applied to an output field in the OUT or INOUT map is generated.

  The valid picture values for COBOL output maps are:

  A B E P S V X Z 0 9 , . + - $ CR DB / and (

  The valid picture values for PL/I output maps are:

  A B E F G H I K M P R S T V X Y and Z

  1 2 3 6 7 8 9 / + - , . * $ CR DB and (

  For PL/I, a currency symbol can be used as a picture character. The symbol can be any sequence of characters enclosed in < and >, for example <DM>.

  Refer to the appropriate language reference manual for the correct syntax of the PICTURE attribute.

  Note: COBOL supports multiple currency signs and multi-character currency signs in PICTURE specifications.

The default currency picture symbol is the dollar sign ($), which represents the national currency symbol; for example the dollar ($), the pound (£), or the yen (•).

The default currency picture symbol may be replaced by a different currency picture symbol that is defined in the SPECIAL NAMES clause. The currency sign represented by the picture symbol is defined in the same clause. For example:

```
SPECIAL NAMES.
  CURRENCY SIGN IS '$' WITH PICTURE SYMBOL '$'.
  CURRENCY SIGN IS '£' WITH PICTURE SYMBOL '£'.
  CURRENCY SIGN IS 'EUR' WITH PICTURE SYMBOL '#'.
WORKING STORAGE SECTION.
    01 USPRICE PIC $99.99.
    01 UKPRICE PIC £99.99.
    01 ECPRICE PIC #99.99.
```

LENGTH must be specified when PICOUT specifies a COBOL picture containing a currency symbol that will be replaced by a currency sign of length greater than 1.

- position is the POS operand. It specifies the location of a field. This operand specifies the individually addressable character location in a map at which the attribute byte that precedes the field is positioned.

  Position is a BMSPositionType that has the following attributes:
  - number specifies the displacement (relative to zero) from the beginning of the map being defined.
  - (line, column) specify lines and columns (relative to one) within the map being defined.

  The location of data on the output medium is also dependent on DFHMDI operands. The first position of a field is reserved for an attribute byte. When supplying data for input mapping from non-3270 devices, the input data must allow space for this attribute byte. Input data must not start in column 1 but may start in column 2.

  The POS operand always contains the location of the first position in a field, which is normally the attribute byte when communicating with the 3270. For the second and subsequent fields of a group, the POS operand points to an assumed attribute-byte position, ahead of the start of the data, even though no actual attribute byte is necessary. If the fields follow on immediately from one another, the POS operand should point to the last character position in the previous field in the group.

  When a position number is specified that represents the last character position in the 3270, two special rules apply:
  - ATTRIB=IC should not be coded. The cursor can be set to location zero by using the CURSOR option of a SEND MAP, SEND CONTROL, or SEND TEXT command.

- If the field is to be used in an output mapping operation with MAP=DATAONLY on the SEND MAP command, an attribute byte for that field must be supplied in the symbolic map data structure by the application program.

- ProgrammedSymbol is the PS operand. It specifies that programmed symbols are to be used. This overrides any PS operand set by the DFHMDI macro or the DFHMSD macro.

  BASE is the default and specifies that the base symbol set is to be used.

  psid specifies a single EBCDIC character, or a hexadecimal code of the form X'nn,' that identifies the set of programmed symbols to be used.

  The PS operand is ignored unless the terminal supports programmed symbols.

  SOSI indicates that the field may contain a mixture of EBCDIC and DBCS data. The DBCS subfields within an EBCDIC field are delimited by SO (shift out) and SI (shift in) characters. SO and SI both occupy a single screen position (normally displayed as a blank). They can be included in any non-DBCS field on output, if they are correctly paired. The terminal user can transmit them inbound if they are already present in the field, but can add them to an EBCDIC field only if the field has the SOSI attribute.

  TRANSP determines whether the background of an alphanumeric field is transparent or opaque, that is, whether an underlying (graphic) presentation space is visible between the characters.

### *BMSFoldType*

BMSFoldType specifies whether to generate lowercase or uppercase characters only in C language programs in the appropriate data structure.

### *BMSHighlightingType*

BMSHighlightingType specifies the default highlighting attribute for all fields in all maps in a mapset. This is overridden by the HILIGHT operand of the DFHMDI, which is in turn overridden by the HILIGHT operand of the DFHMDF. The HILIGHT operand is ignored unless the terminal supports it.

BMSHighlightingType has the following attributes:

- OFF is the default and indicates that no highlighting is used.
- BLINK specifies that the field must blink.
- REVERSE specifies that the character or field is displayed in reverse video, for example, on a 3278, black characters on a green background.
- UNDERLINE specifies that a field is underlined.

### *BMSJustifyType*

BMSJustifyType can be "left," "right," "first," "last," or "bottom."

### *BMSLanguageType*

BMSLanguageType specifies language types:

- Assembler
- C
- COBOL
- COBOL2
- PL/I

### *BMSMap*

BMSMap is implemented by DFHMDI macro. BMSMap has the following attributes:

- MAPNAME is the name of the map and consists of 1-7 characters.
- COLUMN specifies the column in a line at which the map is to be placed, that is, it establishes the left or right map margin.
- JUSTIFY controls whether map and page margin selection and column counting are to be from the left or right side of the page. The columns between the specified map margin and the page margin are not available for subsequent use on the page for any lines included in the map.
- NUMBER is the column from the left or right page margin where the left or right map margin is to be established.
- NEXT indicates that the left or right map margin is to be placed in the next available column from the left or right on the current line.
- SAME indicates that the left or right map margin is to be established in the same column as the last non-header or
- nontrailer map used that specified COLUMN=number and the same JUSTIFY operands as this macro. For input operations, the map is positioned at the extreme left-hand or right-hand side, depending on whether JUSTIFY=LEFT or JUSTIFY=RIGHT has been specified.
- Line is the LINE operand. It specifies the starting line on a page in which data for a map is to be formatted.
  - NUMBER is a value in the range 1-240, specifying a starting line number. A request to map, on a line and column, data that has been formatted in response to a preceding BMS command, causes the current page to be treated as though complete. The new data is formatted at the requested line and column on a new page.
  - NEXT specifies that formatting of data is to begin on the next available completely empty line. If LINE=NEXT is specified in the DFHMDI macro, it is ignored for input operations and LINE=1 is assumed.

- SAME specifies that formatting of data is to begin on the same line as that used for a preceding BMS command. If COLUMN=NEXT is specified, it is ignored for input operations and COLUMN=1 is assumed. If the data does not fit on the same line, it is placed on the next available line that is completely empty.

- SIZE(arg1,arg2) specifies the size of a map. arg2 = line is a value in the range 1-240, specifying the depth of a map as a number of lines. arg1 = column is a value in the range 1-240, specifying the width of a map as a number of columns. This operand is required in the following cases:
  - An associated DFHMDF macro with the POS operand is used.
  - The map is to be referred to in a SEND MAP command with the ACCUM option.
  - The map is to be used when referring to input data from other than a 3270 terminal in a RECEIVE MAP command.

- ShiftOutShiftIn is the SOSI operand. It indicates that the field may contain a mixture of EBCDIC and DBCS data. The DBCS subfields within an EBCDIC field are delimited by SO (shift out) and SI (shift in) characters. SO and SI both occupy a single screen position (normally displayed as a blank). They can be included in any non-DBCS field on output, if they are correctly paired. The terminal user can transmit them inbound if they are already present in the field, but can add them to an EBCDIC field only if the field has the SOSI attribute.

- TioaPrefix is a Boolean type for the TIOAPFX operand. It specifies whether BMS should include a filler in the symbolic description maps to allow for the unused TIOA prefix. This operand overrides the TIOAPFX operand specified for the DFHMSD macro.
  - YES specifies that the filler should be included in the symbolic description maps and should always be used for command-level application programs. If TIOAPFX=YES is specified, all maps within the mapset have the filler. TIOAPFX=YES
  - NO is the default and specifies that the filler is not to be included.

### *BMSMapAttributesType*

BMSMapAttributesType has the following attributes:

- color : Boolean
- highlighting : Boolean
- outline : Boolean
- programmedSymbol : Boolean
- sosi : Boolean
- transparent : Boolean
- validation : Boolean

### *BMSMapset*

BMSMapset is implemented by the DFHMSD macro. BMSMapset has the following attributes:

- type=DSECT ¶ MAP ¶ FINAL. Mandatory, this generates the two bits of a BMS entity.
- mode=OUT ¶ IN ¶ INOUT. OUT is default. INOUT says do both IN and OUT processing. With IN, I is appended to mapname, with OUT, O is appended to mapname.
- lang=ASM¶ COBOL ¶ COBOL2 ¶ PL/l ¶ C. ASM is default.
- fold=LOWER ¶ UPPER. LOWER is default. Only applies to C.
- dsect=ADS ¶ ADSL. ADS is default. ADSL requires lang = C.
- trigraph = YES only applies to lang = C.
- BASE specifies that the same storage base is used for the symbolic description maps from more than one mapset. The same name is specified for each mapset that is to share the same storage base. Because all mapsets with the same base describe the same storage, data related to a previously used mapset may be overwritten when a new mapset is used. Different maps within the same mapset also overlay one another.

  This operand is not valid for assembler-language programs, and cannot be used when STORAGE=AUTO has been specified.
- term = type. Each terminal type is represented by a character. 3270 is default and is a blank. Added to MAPSET name, or, suffix = numchar which is also added to mapset name.
- CURSLOC indicates that for all RECEIVE MAP operations using this map on 3270 terminals, BMS sets a flag in the application data structure element for the field where the cursor is located.
- STORAGE depends upon the language in which application programs are written, as follows:

  For a COBOL program, STORAGE=AUTO specifies that the symbolic description maps in the mapset are to occupy separate (that is, not redefined) areas of storage. This operand is used when the symbolic description maps are copied into the working-storage section and the storage for the separate maps in the mapset is to be used concurrently.

  For a C program, STORAGE=AUTO specifies that the symbolic description maps are to be defined as having the automatic storage class. If STORAGE=AUTO is not specified, they are declared as pointers. You cannot specify both BASE=name and STORAGE=AUTO for the same mapset. If STORAGE=AUTO is specified and TIOAPFX is not, TIOAPFX=YES is assumed.

For a PL/I program, STORAGE=AUTO specifies that the symbolic description maps are to be declared as having the AUTOMATIC storage class. If STORAGE=AUTO is not specified, they are declared as BASED. You cannot specify both BASE=name and STORAGE=AUTO for the same mapset. If STORAGE=AUTO is specified and TIOAPFX is not, TIOAPFX=YES is assumed.

For an assembler-language program, STORAGE=AUTO specifies that individual maps within a mapset are to occupy separate areas of storage instead of overlaying one another.

This is derived from BMSStatement.

### *BMSMapsetType*

BMSMapsetType specifies the type of map to be generated using the definition. Both types of map must be generated before the mapset can be used by an application program. If aligned, symbolic description maps are required. You should ensure that you specify SYSPARM=ADSECT and SYSPARM=AMAP when you assemble the symbolic and physical maps respectively.

BMSMapsetType has the following attributes:

- DSECT specifies that a symbolic description map is to be generated. Symbolic description maps must be copied into the source program before it is translated and compiled.
- MAP specifies that a physical map is to be generated. Physical maps must be assembled or compiled, link-edited, and cataloged in the CICS program library before an application program can use them.
- FINAL denotes the end of a mapset.

### *BMSModeType*

BMSModeType specifies whether the mapset is to be used for input, output, or both (i.e., input and output).

### *BMSOutliningType*

BMSOutliningType is the OUTLINE statement. It allows lines to be included above, below, to the left, or to the right of a field. You can use these lines in any combination to construct boxes around fields or groups of fields.

### *BMSPositionType*

BMSPositionType specifies where on the presentation space the field is to be placed.

### *BMSSizeType*

BMSSizeType has the following attributes:

- line is an integer.
- column is an integer.

### *BMSValidationType*

BMSValidationType is the VALIDN statement. It specifies that validation is to be used if the terminal supports it or this field can be processed by the BMS global user exits.

This overrides any VALIDN operand on the DFHMDI macro or the DFHMSD macro.

BMSValidationType has the following attributes:

- MUSTFILL specifies that the field must be filled completely with data. An attempt to move the cursor from the field before it has been filled, or to transmit data from an incomplete field, raises the INHIBIT INPUT condition
- MUSTENTER specifies that data must be entered into the field, though need not fill it. An attempt to move the cursor from an empty field raises the INHIBIT INPUT condition
- TRIGGER specifies that this field is a trigger field. Trigger fields are discussed in the CICS Application Programming Guide.
- USEREXIT specifies that this field is to be processed by the BMS global user exits, XBMIN and XBMOUT, if this field is received or transmitted in a 3270 datastream when the respective exit is enabled. The USEREXIT specification applies to all 3270 devices.

The MUSTFILL, MUSTENTER, and TRIGGER specifications are valid only for terminals that support the field validation extended attribute, otherwise they are ignored.

# Index

# Index

# *Index*

# UML for Enterprise Application Integration Reference Sheet

The formal version of UML EAI is based on these documents:

- Revised submission document: ad/01-09-17
- FTF Report: ptc/03-10-10
- Convenience document: ptc/03-10-11