Date: June 2010

Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM)

Version 1.2

OMG Document Number: formal/2010-06-03
Standard document URL: http://www.omg.org/spec/KDM/1.2
Associated Schema Files:
ptc/09-05-22 -- http://www.omg.org/spec/KDM/20090501
ptc/09-05-23 -- http://www.omg.org/spec/KDM/20090502
ptc/09-05-24 -- http://www.omg.org/spec/KDM/20090503

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

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

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.

OMG's Issue Reporting Procedure

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

Table of Contents

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

Business Modeling Specifications

- Business Rules and Process Management Specifications

Language Mappings

- IDL/Language Mapping Specifications
- Other Language Mapping Specifications

Middleware Specifications

- CORBA/IIOP
- CORBA Component Model
- Data Distribution
- Specialized CORBA

Modeling and Metadata Specifications

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

Modernization Specifications

- KDM

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

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) All specifications are available in PostScript and PDF format and may be obtained from the Specifications Catalog cited above. Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

OMG Contact Information

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

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

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

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

Helvetica/Arial - 10 pt: Exceptions

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

Issues

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

Roadmap

This roadmap provides a list of documents including all documents that were generated during the revision task force process. The source documents for this specification include:

ptc/2009-06-04: Inventory file

ptc/2009-06-02: RTF Report

ptc/2009-06-03: Convenience document with change bars

ptc/2009-06-05: Convenience document without change bars

ptc/2009-05-22: CMOF XMI

ptc/2009-05-23: EMF XMI

ptc/2009-05-24: KDM XMI Schema

1 Scope

This specification defines a meta-model for representing *existing software*, its elements, associations, and operational environments, referred to as the Knowledge Discovery Meta-model (KDM). This is the first in the series of specifications related to Software Assurance (SwA) and Architecture-Driven Modernization (ADM) activities. KDM facilitates projects that involve *existing software systems* by insuring interoperability and exchange of data between tools provided by different vendors.

One common characteristic of various tools that address SwA and ADM challenge is that they analyze *existing software artifacts* (for example, source code modules, database descriptions, build scripts, etc.) to obtain explicit knowledge. Any tool that operates on existing software produces a portion of the knowledge about existing software system. However, such tool-specific knowledge may not be exported in any explicit format. For example, such knowledge may be used internally by the tool: a compliler generates precise knowledge about a compilation unit only to discard it as soon as the object file is generated. Tool-specific knowledge may be limited in scope, restricted to a particular source language, and/or particular transformation, and/or *operational environment*. All the above may hinder interoperability between different tools. The meta-model for Knowledge Discovery provides a common ontology and an interchange format that facilitates the exchange of data contained within individual tool models that represent *existing software*. The meta-model represents the physical and logical elements of software as well as their relations at various levels of abstraction. The primary purpose of this meta-model is to enable a common interchange format that will allow interoperability between existing modernization and software assurance tools, services, and their respective intermediate representations.

2 Conformance

KDM is defined via Meta-Object Facility (MOF). KDM determines the interchange format via the XML Metadata Interchange (XMI) by applying the standard MOF to XMI mapping to the KDM MOF model. The interchange format defined by KDM is called the KDM XMI schema. The KDM XMI schema is provided as the normative part of this specification.

KDM is a meta-model with a very broad scope that covers a large and diverse set of applications, platforms, and programming languages. Not all of its capabilities are equally applicable to all platforms, applications, or programming languages. The primary goal of KDM is to provide the capability to exchange models between tools and thus facilitate cooperation between tool suppliers to integrate multiple facts about a complex *enterprise application*, as the complexity of modern *enterprise applications* involves multiple platform technologies and programming languages. In order to achieve interoperability and especially the integration of information about different facets of an *enterprise application* from multiple analysis tools, this specification defines several compliance levels thereby increasing the likelihood that two or more compliant tools will support the same or compatible meta-model subsets. This suggests that the meta-model should be structured modularly, following the principle of separation of concerns, with the ability to select only those parts of the meta-model that are of direct interest to a particular tool vendor. Separation of concerns in the design of KDM is embodied in the concept of KDM domains.

2.1 KDM Domains

Separate facts of knowledge discovery in enterprise application in KDM are grouped into several KDM domains (refer to Figure 2.1). Each KDM domain defines an architectural viewpoint. The viewpoint language for the domain is defined by the corresponding KDM package that defines meta-model elements to represent particular facts of the system under study that are essential to the given domain. The meta-model elements defined by all KDM packages constitute the ontology for describing existing software systems. For example, the Code and Action package define the viewpoint language for the

Code domain that represent individual code elements of the system under study, such as variables, procedures, and statements. The Structure packages define the viewpoint language for the Structure domain that represents architectural elements of the same system, such as subsystems and components. The Conceptual package corresponds to the Business Rules domain and defines the viewpoint language to represent behavioral elements of the same system such as features or business rules. KDM formally defines traceability between facts, aggregation and derivation of facts across domains.

The following domains of knowledge have been identified as the foundation for defining compliance in KDM: Inventory, Code, Build, Structure, Data, Business Rules, UI, Event, Platform, and micro KDM.

From the user's perspective, this partitioning of KDM means that they need only to be concerned with those parts of the KDM that they consider necessary for their activities. If those needs change over time, further KDM domains can be added to the user's repertoire as required. Hence, a KDM user does not have to know the full meta-model to use it effectively. In addition, most KDM domains are partitioned into multiple increments, each adding more knowledge capabilities to the previous ones. This fine-grained decomposition of KDM serves to make the KDM easier to learn and use, but the individual segments within this structure do not represent separate compliance points. The latter strategy would lead to an excess of compliance points and result to the interoperability problems described above. Nevertheless, the groupings provided by KDM domains and their increments do serve to simplify the definition of KDM compliance as explained below.

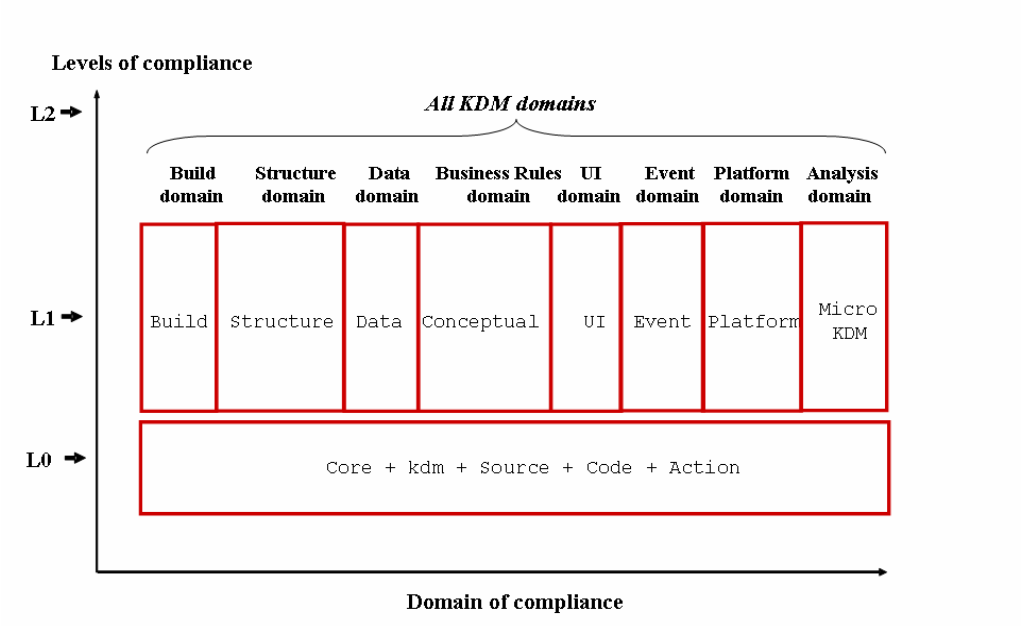

Figure 2.1 - Domains and levels of KDM compliance

2.2 Compliance Levels

In addition, the total set of KDM packages is further partitioned into layers of increasing capability called compliance levels. There are three KDM compliance levels:

- Level 0 (L0) - This compliance level addresses the Inventory and Code domains and is determined by the following KDM packages: Core, kdm, Source, Code, and Action packages. It provides an entry-level of knowledge discovery capability. More importantly, it represents a common denominator that can serve as a basis for interoperability between different categories of KDM tools.

  To be L0 compliant, a tool shall completely support all meta-model elements within all packages for L0 level.

- Level 1 (L1) - This level addresses the remaining KDM domains and extends the capabilities provided by Level 0. Specifically, this level is determined by the following packages: Build, Structure, Data, Conceptual, UI, Event, Platform, as well as the set of constraints for the micro KDM domain defined in sub clause 14 “Micro KDM,” and Annex A “Semantics of the Micro KDM Action Elements.” These packages are grouped to form above-mentioned domains. More importantly, this level represents a layer where tools could be complimentary since their focus would be in different areas of concern. This would be an additional reason why L0 interoperability (which at this level would be viewed as information sharing between tools) is mandated. In this case interoperability at this level would be viewed as correlation between tools to complete knowledge puzzle that end user might need to perform a particular task.

  To be L1 compliant for a given KDM domain, a tool shall completely support all meta-model elements defined by the corresponding packages and satisfy all semantic constraints specified for the domain.

- Level 2 (L2) - This level is the union of L1 levels for all KDM domains.

2.2.1 Meaning and Types of Compliance

Compliance to Level 1 (L1) for a certain KDM domain entails full realization of all KDM packages for the corresponding KDM Domain. This also implies full realization of all KDM packages in all the levels below that level (in this case Level 0 (L0)). It is not meaningful to claim compliance to Level 1 without also being compliant with the Level 0. A tool that is compliant at a Level 1 must be able (at least) to import models from tools that are compliant to Level 0 without loss of information. So, “full realization” for a KDM domain means supporting the complete set of concepts defined for that KDM domain at L1 and complete set of concepts defined at L0.

For a given compliance level, a *KDM implementation* can provide:

- The capability to analyze physical artifacts of existing applications and export their representations based on the XMI schema corresponding to the given compliance level.

- The capability to import representations of existing software systems based on the XMI schema corresponding to the given compliance level and perform operations suggested by the corresponding packages.

Table 2.1 - Compliance Statements

Compliance Statement				
Compliance Level		**Import-Analysis**	**Import API**	**Export**
L0		Compliant tool shall: • Import KDM models based on complete KDM XMI schema into existing tool; • Implement mapping between KDM and existing internal representation of the tool; • Extend operations of existing tool to support meta-model elements of KDM framework; • Extend operations of existing tool to support meta-model elements of Code and Action packages; • Extend operations of existing tool to support traceability to the physical artifacts of the application from Source package.	Compliant tool shall: • Import KDM models based on complete KDM XMI schema; • Support KDM API defined by the KDM Core package; • Support KDM framework as defined in the Kdm package; • Support KDM API defined by the Code and Action packages; • Support traceability to the physical artifacts of the application as defined in the Source package.	Compliant tool shall: • Provide capability to analyze existing artifacts for specified programming language or multiple languages; • Generate XMI documents corresponding to the KDM XMI schema; • Support KDM framework as defined by the Kdm package; • Support Code and Action packages; • Provide traceability back to the physical artifacts as defined by the Source package.
L1	**STRUCTURE**	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Structure package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Structure package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze architecture components of existing application and generate KDM Structure model according to Structure package.
	DATA	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Data package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Data package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze persistent data components of existing application for specified database system and generate KDM Data model according to Data package.
	PLATFORM	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Platform package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Platform and Runtime packages.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze platform artifacts for specified platform and generate KDM Platform model according to Platform package.

Table 2.1 - Compliance Statements

	BUILD	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Build package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Build package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze build artifacts for specified build environment and generate KDM Build model according to Build package.
	UI	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the UI package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the UI package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze user interface artifacts for specified user interface system and generate KDM UI model according to UI package;
	EVENT	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Event package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Event package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze artifacts related to event-driven runtime frameworks and state-transition behavior and generate KDM Event model according to Event package.
	BUSINESS	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support meta-model elements of the Conceptual package.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by the Conceptual package.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze conceptual and behavior artifacts (e.g., domain concepts, business rules, scenarios) of existing application and generate KDM Conceptual model according to Conceptual package;
	MICRO KDM	Compliant tool shall: • Demonstrate L0 compliance for analysis; • Extend operations of existing tool to support micro KDM actions as specified in Chapter 14 micro KDM and Annex A.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support micro KDM actions as specified in Chapter 14 micro KDM and Annex A.	Compliant tool shall: • Demonstrate L0 compliance for export; • Provide capability to analyze artifacts of existing application to the level of detail specified in Chapter 14 and Annex A provide the mapping of semantics of the existing application as it is determined by the programming languages and the runtime platform into KDM micro actions and generate KDM models that represent the same meaning.

Table 2.1 - Compliance Statements

L2	Compliant tool shall: • Demonstrate L0 import compliance for analysis; • Demonstrate L1 import-analysis compliance for all KDM domains.	Compliant tool shall: • Demonstrate L0 compliance for import; • Support KDM API as defined by all KDM packages.	Compliant tool shall: • Demonstrate L0 export compliance; • Demonstrate L1 export compliance for all KDM domains.

3 Normative References

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to or revisions of any of these publications do not apply.

- ISO/IEC 19502:2005 Information technology -- Meta Object Facility (MOF)
- ISO/IEC 19503:2005 Information technology -- XML Metadata Interchange (XMI)
- ISO/IEC 11404:2007 Information technology -- General-Purpose Datatypes (GPD)
- OMG UML Infrastructure Specification, v2.2 formal/2009-02-04
- OMG Meta-Object Facility (MOF), v2.0 formal/2006-01-01
- OMG Semantics of Business Vocabularies and Business Rules (SBVR), v1.0 formal/08-01-02

4 Terms and Definitions

There are no special terms or definitions in this specification.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Changes to Other OMG Specifications

There are no changes to other OMG specifications.

6.2 How to Proceed

The rest of this document contains the technical content of this specification.

Chapter 7. Specification overview - Provides design rationale for the KDM specification.

Chapter 8. KDM - Contains an overview of the KDM packages.

Part I - The KDM Infrastructure Layer

Chapter 9. Core package - Describes foundation constructs for creating and describing meta-model classes in other KDM packages. Classes and associations of the Core package determine the structure of KDM models, provide meta-modeling services to other classes, and define fundamental constraints.

Chapter 10. KDM package - Contains the key infrastructure elements that determine patterns for constructing KDM models and integrating them. This package defines several static elements that are shared by all KDM instances. This package determines the queries against KDM instances.

Chapter 11. Source package - Includes meta-model elements that provide traceability from KDM facts to the original representation of the physical artifact (for example, source code).

Part II - The Program Elements Layer

Chapter 12. Code package - Describes meta-model elements that capture programming artifacts as provided by programming languages, such as data types, procedures, macros, prototypes, templates, etc.

Chapter 13. Action package - Contains the meta-model elements related to the behavior of applications. Action package defines detailed endpoints for most KDM relations. The key element related to behavior is a KDM action. Other packages depend on the Action package to use actions in further modeling aspects of existing applications such as features, scenarios, business rules, etc.

Chapter 14. Micro KDM - Provides the guidelines and constraints for semantically precise KDM representations.

Part III - The Runtime Resources Layer

Chapter 15. Platform package - Describes the meta-model elements that represent operating environments of existing software systems. Application code is not self-contained, as it depends not only on the selected programming language, but also on the selected Runtime platform. Platform elements determine the execution context for the application. Platform package provides meta-model elements to address the following:

- Resources that Runtime platforms provide to components
- Services that are provided by the platform to manage the life-cycle of each resource
- Control-flow between components as it is determined by the platform
- Error handling across application components
- Integration of application components

The Platform package focuses on the logical aspects of the operating environments of existing applications, while the Runtime package further addresses the physical aspects of operating environments, such as deployment.

Chapter 16. UI package - Contains the meta-model elements that represent knowledge related to user interfaces, including their logical composition, sequence of operations, etc.

Chapter 17. Event package - Includes meta-model elements that represent basic elements related to behavior of applications in terms of states, transitions between states, events, messages, and responses.

Chapter 18. Data package - Describes the Data domain of KDM, aiming primarily at databases and other ways of organizing persistent data in enterprise applications independent of a particular technology, vendor and platform.

Part IV - Abstractions Layer

Chapter 19. Structure package - Contains the meta-model elements that represent the logical organization of the software system in terms of logical subsystems, architectural layers, components and packages.

Chapter 20. Conceptual package - Describes the meta-model elements that represent facts related to the business domain of the existing system and provide traceability to other KDM facts.

Chapter 21. Build package - Includes the meta-model elements that represent the facts related to the build process of the software system (including but not limited to the engineering transformations of the “source code” to “executables”).

6.2.1 Diagram Format

Meta-model diagrams in this specification are used to mechanically produce the Meta-Object Facility (MOF) definition of KDM, and the corresponding KDM XMI schema. The following conventions are adopted for all metamodel diagrams throughout this specification:

- An association with one end marked by a navigability arrow means that:
  - the association is navigable in the direction of that end,
  - the marked association end is owned by the classifier, and
  - the opposite (unmarked) association end is owned by the association.
- An association with neither end marked by navigability arrows means that:
  - the association is navigable in both directions,
  - each association end is owned by the classifier at the opposite end (i.e., neither end is owned by the association),
  - additionally, properties “owner,” “group,” and “model” are automatically renamed to ownerProperty, groupProperty, and modelProperty respectively.
- Association specialization and redefinition are indicated by appropriate constraints situated in the proximity of the association ends to which they apply. Thus:
  - the constraint {subsets endA} means that the association end to which this constraint is applied is a specialization of association end endA that is part of the association being specialized.
  - a constraint {redefines endA} means that the association end to which this constraint is applied redefines the association end endA that is part of the association being specialized.
- Derived union is indicated by placing constraint {union} in the proximity of the association end to which it applies. The corresponding association endpoint is marked as derived and read only.
- If an association end is unlabeled, the default name for that end is the name of the class to which the end is attached, modified such that the first letter is a lowercase letter. In addition, if the name of the class to which the end is attached starts has a meaningful prefix of uppercase letters, for example XMLxxxx, KDMxxx, UIxxxx, the entire uppercase prefix is modified to become lowercase. For example, the above words become xmlxxxx, kdmxxx, uixxxx. (Note that, by convention, non-navigable association ends are often left unlabeled since, in general, there is no need to refer to them explicitly either in the text or in formal constraints - although there may be needed for other purposes, such as MOF language bindings that use the metamodel.)

- unlabeled association ends attached to the class KDM Entity that correspond to KDM Relationships are additionally prefixed with “in” or “out” according to the direction of the relationship. The corresponding properties at the KDM Relationship class side are “to” and “from.” For example, association ends for the ActionElement class corresponding to the associations to ControlFlow class are named “inControlFlow” (the counterpart of the “to” endpoint from the ControlFlow side) and “outControlFlow” (the counterpart of the “from” endpoint from the ControlFlow side).

- Associations that are not explicitly named, are given names that are constructed according to the following production rule:

  "A_" <class-name1> "_" <association-end-name2>

  where <class-name1> is the name of the class that owns the first association end and <association-end-name2> is the name of the second association end.

6.3 Acknowledgements

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

- Allen Systems Group, Inc
- BluePhoenix
- EDS
- Flashline
- IBM
- Klocwork, Inc.
- KDM Analytics
- SoftwareRevolution
- Tactical Strategy Group, Inc.
- Unisys

The following persons were members of the core team that designed and wrote this specification: Nikolai Mansourov, Michael Smith, Djenana Campara, Larry Hines, William Ulrich, Howard Hess, Henric Gomez, Chris Caputo, Vitaly Khusidman, Barbara Errickson-Connor.

In addition, the following persons contributed valuable ideas and feedback that significantly improved the content and the quality of this specification: Pete Rivett, Adam Neal, Sumeet Malhotra, Jim Rhyne, Mark Dutra, Sara Porat, Fred Cummins, Manfred Koethe, Alena Laskavaia, Alain Picard.

7 Specification Overview

This specification defines a meta-model for representing information related to existing software, its elements, associations, and operational environments (referred to as the Knowledge Discovery Meta-model (KDM)).

The KDM provides a common interchange format that allows interoperability between existing software analysis and modernization tools, services, and their respective models. More specifically, (KDM) defines a common ontology and an interchange format that facilitates the exchange of data currently contained within individual tool models that represent existing software. The meta-model represents the physical and logical elements of software as well as their relationships at various levels of abstraction.

KDM groups facts about existing systems into several domains each of which corresponds to an ISO 42010 *architectural viewpoint*. Each KDM domain is represented by one or more KDM packages that formalize the *viewpoint language* for the domain. KDM focuses on the entities and their relationships that are essential for the given domain. A KDM representation of a given software system - a KDM instance - is a collection of facts about that system. These facts are organized into KDM models per each domain. KDM model corresponds to an ISO 42010 *architectural view*. KDM facts are further organized into meaningful groups called segments. A KDM segment may include one or more *architectural views* of the given system. KDM instances may be part of the complete architectural description of the system, as defined by ISO 42010, in which case additional requirements of ISO 42010 shall be satisfied by the overall document. KDM instances are represented as XML documents conforming to the KDM XMI schema.

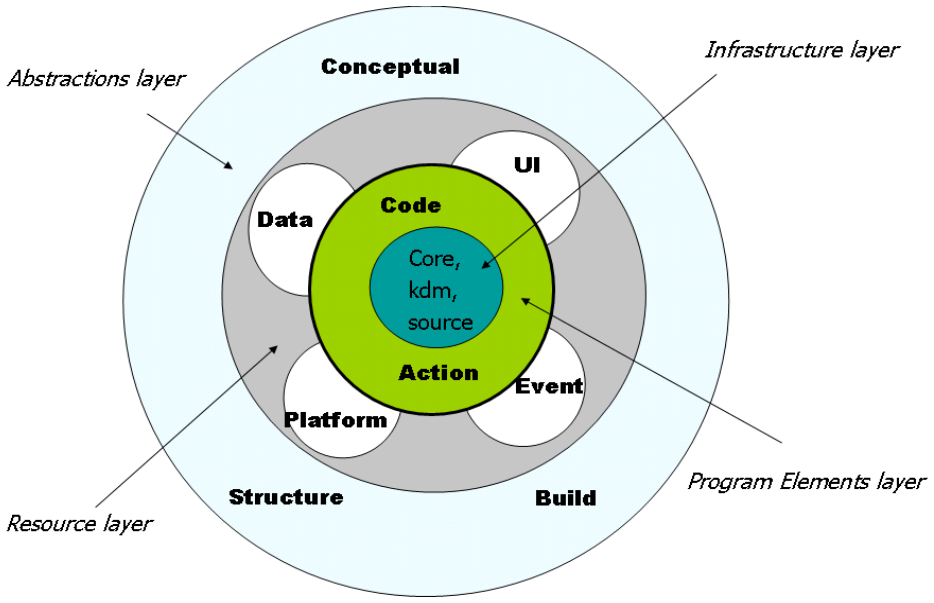

Figure 7.1 - Layers, packages, and separation of concerns in KDM

KDM specification is organized into the following 4 layers:

- KDM Infrastructure Layer
- Program Elements Layer

- Runtime Resource Layer
- Abstractions Layer

Each layer is further organized into packages. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems.

Logically, KDM consists of 9 models. Each KDM model is described by one or more KDM packages and corresponds to one KDM domain. Most KDM domains are defined by a single package, with the exception of the Code domain, which is split between the Code and the Action packages.

The KDM Infrastructure Layer consists of the following 3 packages: Core, kdm, and Source. Core and kdm packages do not describe separate KDM models. Instead these packages define common meta-model elements that constitute the infrastructure for other packages. The Source package defines the Inventory model, which enumerates the artifacts of the existing software system and defines the mechanism of traceability links between the KDM elements and their original representation in the “source code” of the existing software system.

The Program Elements Layer consists of the Code and Action packages. These packages collectively define the Code model that represents the implementation level assets of the existing software system, determined by the programming languages used in the developments of the existing software system. The Code package focuses on the named items from the “source code” and several basic structural relationships between them. The Action package focuses on behavior descriptions and control- and data-flow relationships determined by them. The Action package is extended by other KDM packages to describe higher-level behavior abstractions that are key elements of knowledge about existing software systems.

The Runtime Resource Layer consists of the following 4 packages: Platform, UI, Event, and Data.

The Abstractions Layer consists of the following 3 packages: Structure, Conceptual, and Build.

Each of these knowledge facets contains large amounts of information, impossible to be processed at once by human beings. To overcome such a roadblock, each dimension supports the capability to aggregate (summarize) information to different levels of abstraction. This requires KDM to be scalable. In addition, KDM represents both kinds of information: primary and aggregate information. Primary information is assumed to be automatically extracted from the source code and other artifacts, including (but not restricted to) formal models, build scripts, configuration files, data definition files. Some (or even all) primary information can be provided manually by analysts and experts. Aggregate information is obtained from primary information.

Knowledge Discovery exists at progressively deeper levels of understanding, reflecting varying levels required to achieve different objectives. These were seen as the lexical or syntactic understanding of the program code (language-dependent level); the understanding of the application functionality and design (language-independent level); understanding of application packaging and the corresponding dependencies (architecture level); and an understanding of the applications behavior (business level).

The following are key design characteristics of KDM:

- KDM is a Meta-Object Facility (MOF) model.
- KDM is an Entity-Relationship model.
- KDM can be extended to represent language-specific, application-specific, and implementation-specific entities and relationships.

- KDM models are composable (it is possible to group several entities into a typed container, that will further on represent the entire collection of grouped entities via aggregated relationships). KDM defines multiple hierarchies of entities via containers and groups.
- Analyst is able to refactor the model (for example, by moving entities between containers) and map changes in the model to changes in the software through traceability links.
- KDM is aligned with ISO/IEC 11404 General-Purpose Datatypes and OMG Semantics of Business Vocabularies and Business Rules (SBVR).
- KDM defines an ontology for describing existing software systems. The ontology defined by KDM is related to the elements of existing software systems, the relationships between these elements, as well as the elements of the operational environment of the software system. KDM ontology addresses both physical elements (for example, a procedure, a variable, a table), which are originally represented by language-specific artifacts of the software (for example source code), as well as logical elements (for example, user interface elements, concepts that are implemented by the software, architectural components of the software, such as layers, etc.).

8 KDM

8.1 Overview

KDM specifies a comprehensive set of common concepts required for understanding existing software systems in preparation for software assurance and modernization and provides infrastructure to support specialized definitions of domain-specific, application-specific, or implementation-specific knowledge.

The structure of KDM is defined by combining dimensions and levels of Knowledge Discovery (refer to Figure 8.1).

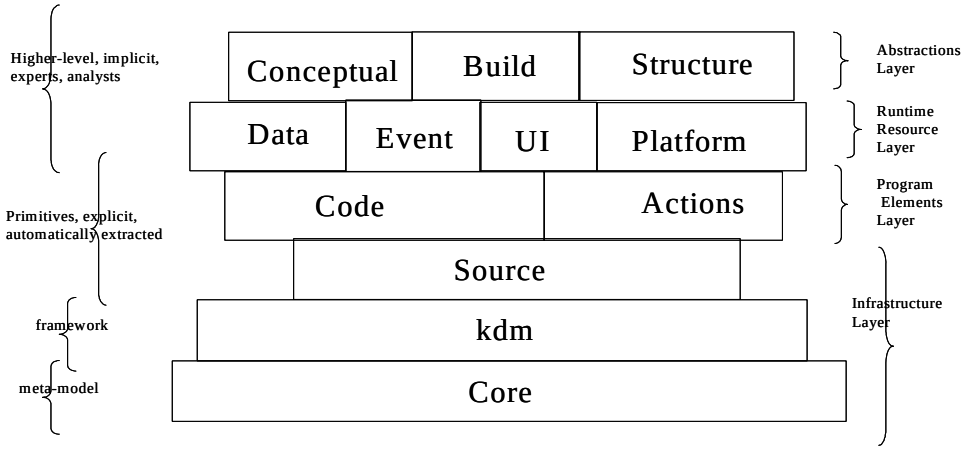

Figure 8.1 - Structure of KDM Packages

The KDM contains 12 packages; each package is defined by one or more class diagrams. Core KDM package defines the basic meta-classes (entity, relationship, container hierarchies, etc.) and well-formedness rules of KDM models.

Figure 8.1 illustrates the layers of the KDM specification and shows dependencies between KDM packages by arranging packages into a stack. Each package depends on one or more packages at the lower layers of the stack. In particular, each KDM package depends on the Core package. The nature of this dependency is that the meta-elements defined by each package are subclasses of one of the core classes. Also, each package depends on the kdm package. Each KDM package above the kdm package defines a KDM model, which corresponds to a certain facet of knowledge about an existing software system. The kdm package provides the infrastructure for all KDM models. The nature of the dependency on the kdm package is twofold. First, each package defines a subclass of the KDMModel class, defined in the kdm package. Second, each kdm package provides several concrete classes that are instantiated in each KDM representation as part of the infrastructure. kdm package defines several important mechanisms that are used by all KDM models: the annotation mechanism, the mechanism of user-defined attributes, and the light-weight extension mechanism. The meta-model elements that support these mechanisms can be instantiated by any KDM model.

The Source package and the Code package of the Program Elements Layer represent the most fundamental, primitive knowledge about existing software systems. Most pieces of this knowledge are explicitly represented by the original source code of the existing software system. It is expected that KDM implementations extract this kind of knowledge automatically, for example by implementing a bridge to an existing software development environment. Such bridge provides a *mapping* from the programming language (or languages) used for the development of the existing software system, to a language-independent KDM representation, that can be further analyzed and transformed by various KDM tools.

Packages of the Runtime Resource Layer represent higher-level knowledge about existing software systems. Most pieces of this knowledge are implicitly represented by the original source code of the software system and the corresponding configuration and resource descriptions. This kind of knowledge is determined not by the syntax and semantics of the programming language (or languages) used for the development of the existing software system, but by the corresponding runtime platform. Incremental analysis of the primitive KDM representation may be required to extract and explicitly represent some of these pieces of knowledge. KDM implementations of the corresponding packages define a mapping from the platform-specific artifacts to a language- and platform-independent KDM representation, that can be further analyzed and transformed by various KDM tools.

Packages of the Abstractions Layer represent even higher-level abstractions about existing software, such as domain-specific knowledge, business rules, implemented by the existing software system, architectural knowledge about the existing software system, etc. This knowledge is implicit, and often there is no formal representation of such knowledge anywhere in the artifacts of the existing software system (and often, even in the documentation). Extracting this kind of knowledge and part of the integrated KDM representation usually involves input from experts and analysts.

KDM instance is a single, integrated representation of different facets of knowledge about the software system.

8.2 Organization of the KDM Packages

KDM defines a collection of meta-model elements whose purpose is to represent existing software artifacts as entities and relations. The KDM has the following organization:

- The Core package defines the basic abstractions of KDM.
- The Kdm package provides static context shared by all KDM models.
- The Source package defines meta-model elements that represent the inventory of the physical artifacts of the existing software system and defines the key traceability mechanism of KDM - how KDM facts references back to their original representation in the artifacts of the software system (for example, source code).
- The Code package defines meta-model elements that represent the low-level building blocks of software, such as procedures, datatypes, classes, variables, etc. (as determined by a programming language).
- Action package defines meta-model elements that represent statements as the end points of relations, and the majority of low-level KDM relations.
- Platform package defines meta-model elements that represent the run time resources used by the software system, as well as relationships determined by the run-time platform.
- UI package defines the meta-model elements that represent the user-interface aspects of the software system.
- Event package defines meta-model elements that represent event-driven aspects of the software system, such as events, states, state transitions, as well as relationships determined by the event-driven semantics of the run-time framework.
- Data package defines meta-model elements that represent persistent data aspects of the software system.
- Structure package defines meta-model elements that represent architectural components of existing software systems, such as subsystems, layers, packages, etc. and define traceability of these elements to other KDM facts for the same system.
- Conceptual package defines meta-model elements that represent the domain-specific elements of the software system.
- Build package defines meta-model elements that represent the artifacts related to the build process of the software system (including but not limited to the engineering transformations of the “source code” to “executables”).

Part I - KDM Infrastructure Layer

KDM is a large specification, since it provides an intermediate representation for several facets of knowledge about existing enterprise software systems. In order to manage the complexity of KDM, a small set of concepts was selected and systematically used throughout the entire specification. These concepts are defined in the so-called KDM Infrastructure Layer. It consists of the following 3 packages:

- Core
- kdm
- Source

The Core package defines the fundamental meta-model element types and the corresponding constraints. Core package provides a set of types that determine each individual KDM meta-model element through subclassing. Each KDM meta-model element is a subclass of one of the core classes. From the meta-model perspective KDM is an entity-relationship representation. So, the two fundamental classes of the Core package are KDMEntity and KDMRelationship. An entity is a thing of significance, about which information needs to be known or held. A KDM entity is an abstraction of some element of an existing software system, that has a distinct, separate existence, a self-contained piece of data that can be referenced as a unit. Each KDM package defines several meta-model elements that represent specific entities for a particular KDM domain.

A relationship represents an association, linkage, or connection between entities that describes their interaction, the dependence of one upon the other, or their mutual interdependence. A KDM relationship represents some semantic association between elements of an existing software system. Each KDM package defines several meta-model elements that represent specific relationships for a particular KDM domain. All KDM relationships are binary.

KDM defines two special relationships:

- containment
- grouping

Some KDM entities are *containers* for other entities. There is a special *container ownership* (containment) relationship between a container and the entities that are directly owned by this container. Some KDM entities are *groups* of other KDM entities. There is a special *group association* (grouping) relationship between the group and the entities that are directly “grouped into” this group.

Core package defines an analysis mechanism, the so-called Aggregated Relations mechanism that brings together special relationships of containment and grouping and regular relationships of the entity-relationship model.

Core package defines a reflective API to KDM representation. Other KDM packages extend this API by specific operations, corresponding to specific facets of knowledge about existing software systems.

Small KDM Core is aligned with the OMG SBVR specification, as it provides an abstraction of existing software systems in the form of *terms* (various KDM entities) and *facts* (various attributes of KDM entities, and relationships between KDM entities). Indeed, most of the KDM specification is a definition of a language- and platform-independent ontology of existing software systems. This alignment is important since KDM can be viewed as a standard vocabulary related to descriptions of existing software systems. SBVR rules can be written using this vocabulary to formally describe further properties of existing software systems.

The kdm package defines several elements that together constitute the *framework* of each KDM representation. The framework determines the physical structure of a KDM representation. The elements of the framework are present in every instance of KDM that represents a particular existing software system. From the framework perspective, each KDM representation consists of one or more Segments, where each Segment owns several KDM models. Each KDM package defines some specific type of KDM model, which addresses a certain specific facet of knowledge about existing software systems. Individual KDM implementations may support one or more selected KDM models, as defined in the KDM compliance section. KDM tools may use multiple KDM implementations to represent different facets of knowledge about the existing software system and integrate them into a single coherent representation. Further, KDM designs facilitate incremental implementations, where certain pieces of knowledge about the existing software is collected by analyzing more lower level KDM representations. According to this approach certain KDM tools may perform a "KDM enrichment" process, a "KDM to KDM transformation," where a tool analyzes the input KDM model and produces one or more additional Segments and Models, explicitly representing certain derived pieces of knowledge about the system.

The Source package defines the so called Inventory model, which represents the physical artifacts of an existing system as KDM entities as well as the mechanism of *traceability links* that provide associations between KDM elements and their "original" language-dependent representation in the source code of the existing software system, for which the KDM representation is created. This is an important part of the KDM Infrastructure, because other KDM packages use this mechanism to refer to the source code and the physical artifacts of the existing software system.

9 Core Package

9.1 Overview

The Core package provides basic constructs for creating and describing meta-model classes in all specific KDM packages. Classes of the Core package determine the structure of KDM models, define fundamental modeling constraints, and determine the reflective API to KDM instances.

9.2 Organization of the Core Package

The KDM uses packages to control complexity and bring together logically interrelated classes. The Core package defines a set of meta-model elements that have the purpose of defining the fundamental patterns and constraints implemented by all other KDM packages.

The KDM Core package consists of the following four class diagrams:

- CoreEntities
- CoreRelations
- AggregatedRelations
- Datatypes

The Core package depends on no other packages.

9.3 CoreEntities Class Diagram

The CoreEntity class diagram defines key abstractions shared by all KDM models. The classes and associations of the CoreEntities class diagram are shown in Figure 9.1.

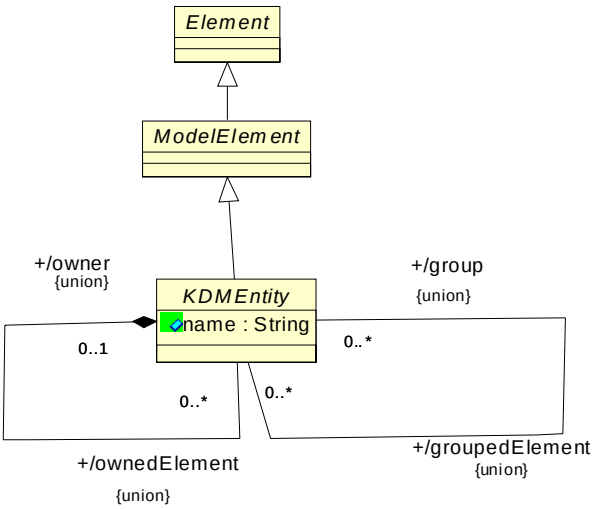

Figure 9.1 - CoreEntities Class Diagram

9.3.1 Element Class (abstract)

An element is an atomic constituent of a model. In the meta-model, an Element is the top meta-element in the KDM class hierarchy. Element is an abstract meta-model element.

Semantics

Element is the common parent from all meta-model elements of KDM. Most subclasses of Element can own *annotations* and user-defined *attributes* through mechanisms defined in the kdm package.

9.3.2 ModelElement Class (abstract)

A model element is an element that represents some aspect of the existing system.

In the meta-model, a ModelElement is the base for all meta-elements of KDM. All other meta-elements are either direct or indirect subclasses of ModelElement. ModelElement is an abstract meta-model element.

A ModelElement can be extended through the *lightweight extension mechanism*.

Superclass

Element

Semantics

The ModelElement is a common class for all meta-model elements that represent some element of the existing software system. The subclasses of Element that are not the subclasses of the ModelElement class are the auxiliary elements of the KDM infrastructure.

Each subclass of the ModelElement meta-model element can be extended through the *light-weight extension mechanism* defined in the kdm package.

9.3.3 KDMEntity Class (abstract)

A KDM entity is a named model element that represents an artifact of existing software systems.

In the meta-model, KDMEntity is a subclass of ModelElement. Each KDM package defines specific KDM entities that are direct or indirect subclasses of KDMEntity. A KDMEntity can be either an *atomic element,* a *container* for some KDMEntities, or a *group* of some KDMEntities. Container and group introduce *implicit relationships between entities* and are used to represent hierarchies of entities. A container is a KDMEntity that owns other entities. A group is a KDMEntity with which other entities are associated. A KDMEntity can be owned by at most one container, and can be associated with zero or many groups.

Superclass

ModelElement

Attributes

name: String	An identifier for the KDM entity.

Associations

owner:KDMEntity[0..1]	KDM *entity* that owns the current element. This property determines a meta-level interface to KDM entities. This property is a derived union. Some KDM entities define a concrete set of owned elements that are subtypes of KDMEntity. In KDM this is represented by the CMOF “derived union” mechanism. Concrete properties subset the “union” properties of the parent classes, defined in the Core package. The owner of a KDM entity is defined as the container for which the given entity is an owned entity.
group:KDMEntity[0..*]	Set of KDM *entities* with which the current element is associated. This property determines a meta-level interface to KDM entities. This property is a derived union. Some KDM entities define a concrete set of grouped elements that are the subtypes of KDMEntity. In KDM this is represented by the CMOF “derived union” mechanism. Concrete properties subset the “union” properties of the parent classes, defined in the Core package. The group of a KDM entity is defined as the group for which the given entity is a grouped entity. Each KDM entity can be associated with multiple groups.

Constraints

1. KDMEntity can have either the ownedElement or groupedElement, but not both.
2. KDMEntity should not reference self as groupedElement.

Operations

getOwner(): KDMEntity[0..1]	This operation returns the KDM entity that is the owner of the current KDM Entity. The owner entity is a KDM container. There can be at most one owner for each given entity.
getOwnedElement():KDMEntity[0..*]	This operation returns the set of KDM entities that are owned by the current KDM Entity. Only KDM containers can own other entities.

getGroup():KDMEntity[0..*]	This operation returns the set of KDM Entities that have a group association to the current KDM Entity. The group entity is a KDM group. Unlike KDM containers, there may be many groups that have an association to a given entity.
getGroupedElement():KDMEntity[0..*]	This operation returns the set of KDM entities that are “grouped” by the current KDM entity. Only KDM groups can have group associations to other entities.

Semantics

An entity is a thing of significance, about which information needs to be known or held. A KDM entity is an abstraction of some element of an existing software system, that has a distinct, separate existence, a self-contained piece of data that can be referenced as a unit. Each KDM package defines several meta-model elements that represent specific entities for a particular KDM domain.

9.4 CoreRelations Class Diagram

The Core class diagram defines key meta-model associations of KDM models. The classes and associations of the CoreRelations class diagram are shown in Figure 9.2.

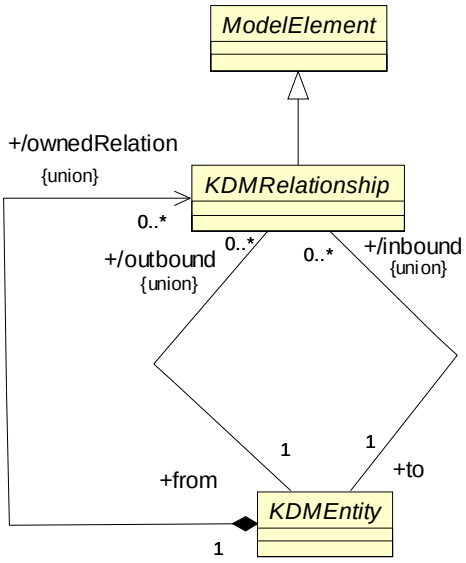

Figure 9.2 - CoreRelations Class Diagram

9.4.1 KDMRelationship Class (abstract)

A KDM relationship is a model element that represents semantic association between two entities.

In the meta-model, KDMRelationship is a subclass of ModelElement. Each KDM package defines some specific KDM relations that are either direct or indirect subclasses of KDMRelationship. Specific subclasses of KDMRelationship are typed associations between some specific subclasses of KDMEntity.

Superclass

ModelElement

Associations

to: KDMEntity[1]	The *target* entity (also referred to as the to-endpoint of the relationship). This property determines a meta-level interface to KDM relationships. Every specific KDM relationship redefines the to-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF “redefines” mechanism. Concrete properties redefine the properties of the parent classes, defined in the Core package.
from:KDMEntity[1]	The *origin* entity (also referred to as the from-endpoint of the relationship). This property determines a meta-level interface to KDM relationships. Every specific KDM relationship redefines the from-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF “redefines” mechanism. Concrete properties redefine the properties of the parent classes, defined in the Core package.

Operations

getTo(): KDMEntity[1]	This operation returns the KDM entity that is the to-endpoint (the target) of the current relationship
getFrom():KDMEntity[1]	This operation returns the KDM entity that is the from-endpoint (the origin) of the current relationship.

Semantics

A relationship represents an association, linkage, or connection between entities that describes their interaction, the dependence of one upon the other, or their mutual interdependence. A KDM relationship represents some semantic association between elements of an existing software system. Each KDM package defines several meta-model elements that represent specific relationships for a particular KDM domain. All KDM relationships are binary.

9.4.2 KDMEntity (additional properties)

Associations

ownedRelation: KDMRelationship[0..*]	Primitive KDM relationships that originate from the current entity.

Operations

getInbound(): KDMRelationship[0..*]	This operation returns the set of relationships such that the current KDMEntity is the to-endpoint of these relations.
getOutbound():KDMRelationship[0..*]	This operation returns the set of relationships such that the current KDMEntity is the from-endpoint of these relationships.

getOwnedRelation():KDMRelationship[0..*]	This operation returns the set of relationships such that the current KDMEntity owns these relationships.

Constraints

1. The set of ownedRelations for a given KDMEntity should be the same as the set of KDMRelations for which the from property is the given KDMEntity.

Semantics

This property defines the so called “encapsulated relationship” pattern. From the infrastructure perspective, the ownedRelation association is required to manage relationship elements. KDM relationships are owned by the entity, which is the origin of the relationship. From the meta-model perspective, each relationship is a self-contained association class with to- and from- properties.

9.5 AggregatedRelations Class Diagram

The AggregatedRelations class diagram defines the key analysis mechanism of KDM. AggregatedRelationships are part of the “meta-level” interface to KDM models, along with interfaces defined by KDMEntity and KDMRelationship.

Overall management and lifecycle of the Aggregated Relationships is determined by the operations of the KDMEntity class.

The classes and associations of the AggregatedRelations class diagram are shown in Figure 9.3.

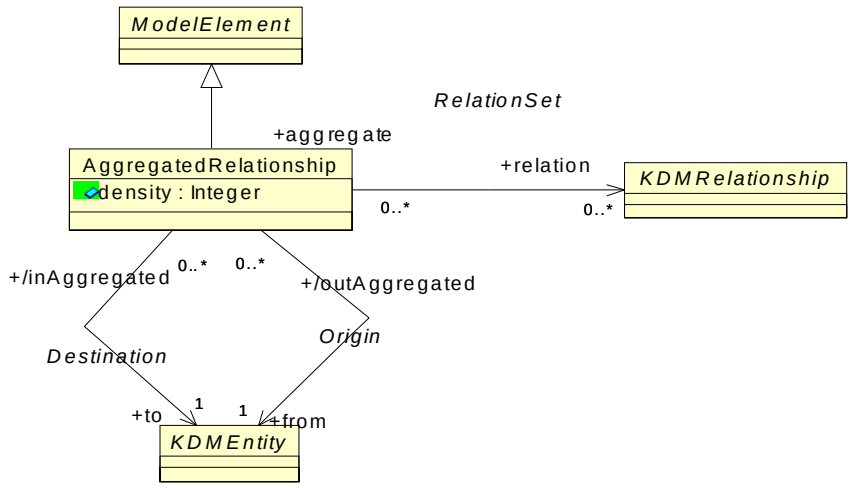

Figure 9.3 - AggregatedRelations Class Diagram

9.5.1 AggregatedRelationship Class

The set of aggregated relationships for a given entity represents all primitive relationships between the entities that are transitively owned by the given entity as well as the entity itself. This is a concrete class, because an AggregatedRelationship can be instantiated, and exchanged. AggregatedRelations are meant to be built on demand (and exchanged too, if necessary). The lifecycle of the Aggregated Relationships can be explicitly managed by the operations of the KDMEntity class.

Superclass

ModelElement

Attributes

density:Integer	The number of primitive relationships in the aggregated set.

Associations

relation:KDMRelationship[0..*]	The set of primitive KDM relationships represented by the aggregated relationship.
to: KDMEntity[1]	The target container of the relationships in the aggregated set. All relations in the aggregated set should terminate at the target container or at some entity that is contained directly or indirectly in the target container.
from:KDMEntity[1]	The source container of the relationships in the aggregated set. All relationships in the aggregated set should originate from the source container or from some entity that is contained directly or indirectly in the source container.

Constraints

1. To- and from-endpoints should be distinct.
2. The density should be greater than or equal to 1.
3. The density should be the same as the number of primitive relations represented by the given aggregated relationship.

Semantics

AggregatedRelations are determined by how atomic elements are owned by containers (or referenced by groups) in the following way:

1. AggregatedRelationship between two entities (no owned elements) represents the set of regular KDM relationships between these two entities (such that the first entity is the from-endpoint of the relationship, and the second entity is the to-endpoint of the relationship).
2. AggregatedRelationship between an entity and a container (or group) represents the set of all regular KDM relationships such that the given entity is the from-endpoint and the to-endpoint is any entity that is owned by the given container (directly or indirectly).
3. AggregatedRelationship between a container (or group) and an entity represents the set of all regular relationships such that the to-endpoint is the given entity and the from-endpoint is any entity that is owned by the given container (directly or indirectly).

4. AggregatedRelation between two containers represents the set of all regular KDM relations such that the from-endpoint is an entity owned by the first container and the to-endpoint is an entity owned by the other container.

A regular KDM relationship is represented by a subtype of KDMRelationship class. It has a concrete type, and an implied density of 1. An AggregatedRelationship represents a set of regular KDM relationships. It has density of greater or equal than 1 and no concrete type (as it may represent regular KDM relationships of different types). An AggregatedRelationship cannot be constructed between two entities if there are no regular KDM relationships between them (according to the definition above).

The relation “x in* C” means that x is in container C or in some sub-container of C, transitively.

For relation R, let R’ be the corresponding aggregated relation.

Given containers C1 and C2 and the relation R, let

P = {(x,y) : x in* C1 and y in* C2 and x R y}

That is, P is the set of pairs such that x is in* C1 and y is in* C2 and x R y.

Then

C1 R’ C2 iff |P| > 0

C1 and C2 are related by the aggregated relation R’ if and only if there is at least one pair in the set P.

The density of C1 ‘' C2 is then simply |P|, the size of the set P.

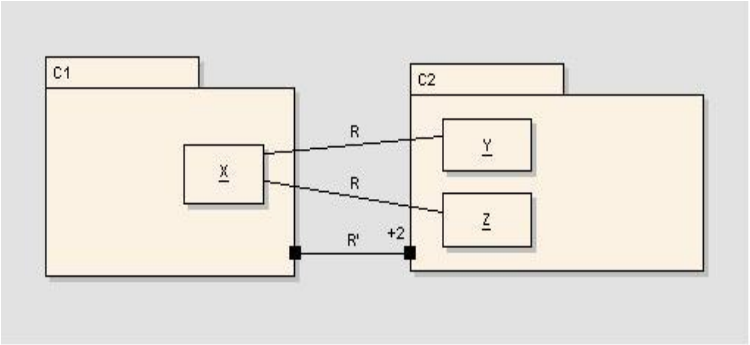

Figure 9.4 - AggregatedRelations illustrated

Figure 9.4 illustrates Containers and aggregated relations. It uses the following notation. UML package symbols represent KDM Containers, UML associations represent KDMRelations. An arrow at the end of an association indicates the direction of the relation (there should be at least one arrow; arrows at both ends indicate two relations, one in each direction). Numbers at the ends of associations represent the density of the corresponding KDM relationship. The KDM density has a different interpretation that UML multiplicity: since KDM represents an existing application, the exact relations and their number is what the model captures. KDM model is not a model that represents constraints, like the ones used during the design phase, rather, this is a model that captures precise knowledge about the application. So, the KDM densities are exact.

Aggregated relations are collections of more primitive relations, which at the end are some basic code fact, for example “procedure x calls procedure y.” Such basic fact has density 1. The primitive code relation represents some basic fact about the existing application. Now, when there are two or more such facts, for example “procedure x in module A calls

procedure y in module B" and "procedure z in module A calls procedure y in module B," there is an aggregated relation between modules A and B with density 2 (2=1+1). In this case, the aggregated relation represents the collection of the two primitive relations between modules A and B.

9.5.2 KDMEntity (additional properties)

Operations

createAggregation(otherEntity:KDMEntity)	This operation creates an aggregated relationship such that the current entity is the from-endpoint of the aggregated relation and the "otherEntity" is the to-endpoint. The new aggregated relationship is owned by the model to which owns the current entity (either directly or indirectly through container ownership).
deleteAggregation (aggregatedRelation:AggregatedRelationship)	This operation deletes the given aggregated relationship.
getInAggregated():AggregatedRelationship[0..*]	This operation returns the set of AggregatedRelationship for which the target is the current KDM Entity.
getOutAggregated():AggregatedRelationship[0..*]	This operation returns the set of AggregatedRelationship for which the origin is the current KDM Entity.

9.6 Datatypes Class Diagram

The Datatypes class diagram collects together utility data types for the Core package. Each class at the Datatypes class diagram is a subclass of MOF DataType class. The classes of the Datatypes class diagram are shown in Figure 9.5.

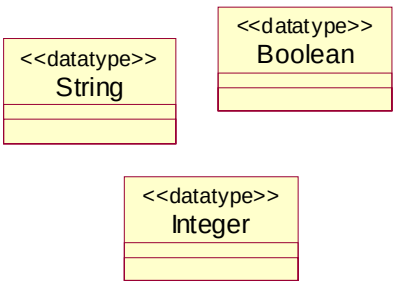

Figure 9.5 - Datatypes Class Diagram

9.6.1 Boolean Type (datatype)

The meta-model uses the Boolean type to represent some KDM attributes, KDM operations, and their parameters.

9.6.2 String Type (datatype)

The meta-model uses the String type to represent some KDM attributes, KDM operations, and their parameters.

9.6.3 Integer Type (datatype)

The meta-model uses the Integer type to represent some KDM attributes, KDM operations, and their parameters.

10 KDM Package

10.1 Overview

The Kdm package defines key infrastructure elements that determine patterns for constructing KDM representations of existing software systems. KDM representations (also referred to as KDM instances) are instances of the KDM (which is a meta-model), where each KDM element represents a certain element of the existing system. Although in the technical sense, KDM instance is a model of the corresponding existing software system, KDM instance is not a model that represents constraints, like the ones used during the design phase, rather, this is an *intermediate representation* that captures precise knowledge about the system.

Implementers of KDM tools are guided by a mapping from the elements of programming languages, runtime platforms, and other artifacts of existing software systems into KDM elements, using semantic description and implementer's guidelines of this specification. Kdm package describes several infrastructure elements that are present in each KDM instance. Together with the elements defined in the Core package these elements constitute the so-called *KDM framework*. The remaining KDM packages provide meta-model elements that represent various elements of existing systems.

Each KDM package follows a uniform meta-model pattern for extending the KDM framework (the Framework Extension meta-model pattern). KDM Framework is part of the KDM Infrastructure together with the elements defined in the Source package.

10.2 Organization of the kdm Package

The kdm package is a collection of classes and associations that define the overall structure of KDM instances. From the infrastructure perspective, KDM instances are organized into *segments* and then further into specific *models*. There are 9 kinds of models. Each KDM model is described by one or more KDM packages and corresponds to one KDM domain. From the architectural perspective, each KDM package defines an *architectural viewpoint.* KDM model is the key mechanism to organize individual facts into *architecture views*. From the infrastructure perspective, a KDM model is a typed container for meta-model element instances (collection of facts organzied into an *architectural view).* From the meta-model perspective, each KDM model is represented by a separate KDM package that defines a collection of the meta-model elements, which can be used to represent the facts about the given existing systems from a particular *viewpoint*. KDM framework defines a common superclass model element for all models - the KDMModel class. KDM specification uses the term "KDM model" to refer both to a meta-model element corresponding to a particular model kind and to a particular instance of such element in a concrete representation of some existing system. Explicit disambiguation (KDM model meta-model element vs. KDM model instance) will be provided when necessary.

KDM model is the key unit of a KDM instance. KDM segment can own one or more models. A segment is a minimal unit of exchange in the KDM ecosystem. Segments can be nested.

The implementer shall provide an adequate partitioning of the KDM instance into multiple models and segments.

A segment is a coherent collection of one or more related models that represents a self-contained perspective on the artifacts of the existing system. It is expected that a complete segment is extracted as a unit. It is expected that each model segment describe artifacts that involve a single programming language and a single platform.

An enterprise application may involve multiple segments that are exported by separate extractor tools, and may need to be integrated to provide a coherent holistic view.

The Kdm package consists of the following 5 class diagrams:

1. Framework – defines the basic elements of the kdm framework.
2. Audit – defines audit information for KDM model elements.
3. Annotations – provides user-defined attributes and annotations to the modeling elements.
4. Extensions – defines the overall organization of the light-weight extension mechanism of KDM.
5. ExtendedValues – the tagged values used by the light-weight extension mechanism.

The Kdm package depends only on the Core package.

10.3 Framework Class Diagram

The Framework class diagram defines the meta-model elements that constitute the so-called KDM Framework: a collection of KDM models organized into nested segments. These meta-model elements determine the structure of KDM instances. The classes and association of the Framework diagram are shown in Figure 10.1.

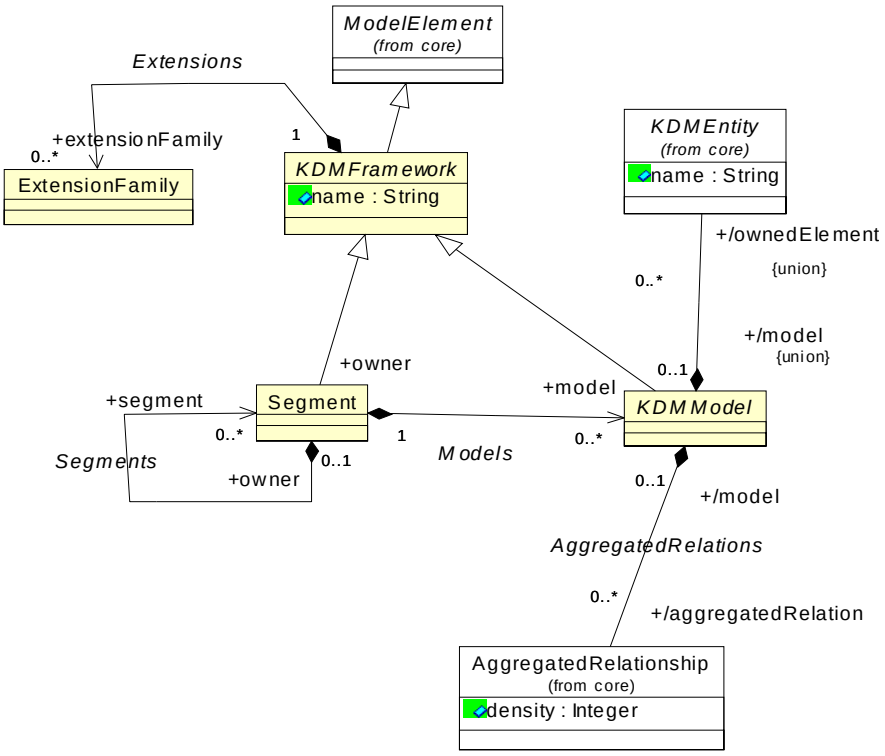

Figure 10.1 - Framework Class Diagram

10.3.1 KDMFramework Class (abstract)

The KDMFramework meta-model element is an abstract class that describes the common properties of all KDM Framework elements. KDMFramework class is extended by Segment and KDMModel classes. Each KDM Framework is the container for KDM light-weight extensions (extension property). The KDM extension mechanism is described further in this chapter.

Superclass

ModelElement

Attributes

name: String [0..*]	The name of the framework element.

Associations

extension: ExtensionFamily [0..*]	Extensions for the current model segment.

Semantics

Concrete instances of the KDM Framework meta-model elements define the organization of the KDM instance. The implementer shall:

- arrange instances of the KDM model elements into models (constrained only by the definition of each model).
- arrange KDM models into one or more segments.
- provide names to KDM models and KDM segments.

The implementers of KDM import tools should not make any assumptions about the organization of KDM instances.

10.3.2 KDMModel Class (abstract)

A KDMModel is an abstract class that defines common properties of KDM model instances that are collections of facts about a given software system from the same *architectural viewpoint* of one of the KDM domains. KDM defines several concrete subclasses of the KDMModel class, each of which defines a particular kind of a KDM model. The *architectural viewpoint* is defined by the corresponding KDM package. A KDM model instance is an *architectural view* of the given system.

From the meta-model perspective, KDMModel extends the Element class. Each concrete KDM model follows the so-called Framework Extension meta-model pattern. This pattern involves the following naming conventions. Let's assume that "foo" is the name of the KDM model. The following rules describe the Framework Extension meta-model pattern:

- The meta-model elements for KDM model "foo" are described in a separate package, called "foo."
- The package defines a concrete subclass of the KDMModel, called "FooModel."
- The package defines a common abstract parent for all KDM entities specific to this KDM model, called "AbstractFooElement." This class extends the KDMEntity class from the Core package.

- The package defines a common abstract parent for all KDM relationship specific to this KDM model, called "AbstractFooRelationship." This class extends KDMRelationship class from the Core package.
- Class "FooModel" owns class "AbstractFooElement." This association subsets the association between the KDMModel and KDMEntity defined at the Framework class diagram.
- Class "AbstractFooElement" owns zero or more AbstractFooRelationship elements.
- The package "foo" includes a "FooInheritances" class diagram, describing inheritances of "FooModel," "AbstractFooElement," and "AbstractFooRelationship" classes, as well as any other common properties related to the KDM Infrastructure, such as properties related to the Source package.
- The package "foo" includes "ExtendedFooElements" diagram that defines two generic meta-model elements "FooElement" and "FooRelationship." These meta-model elements are extension points to package "foo."

Superclass

KDMFramework

Associations

ownedElement: KDMEntity[0..*]	Instances of KDM entities owned by the model. Each KDM model defines specific subclasses of KDMEntity class.
aggregatedRelation:AggregatedRelationship[0..*]	Instances of KDM aggregated relations owned by the model.

Semantics

The implementer shall arrange instances of the KDM model elements into models (constrained only by the definition of each model) and to provide name attributes for each KDM model instance. A KDM model instance may be empty or may contain one or more instances of the elements allowed for this KDM model. A particular KDM instance may contain no KDM models of a given kind, or several such models.

In general, KDM does not constrain associations between instances across KDM models or across KDM segments.

Usually, KDM models corresponding to the KDM Resource Layer and Abstractions Layer have associations to the models in KDM Program Elements and Infrastructure layer. There should be no associations from the Program Elements and Infrastructure layer models to Resource and Abstractions layer models.

The implementers of KDM import tools should not make any assumptions regarding the organization of the content into KDM models.

10.3.3 KDMEntity (additional properties)

Operations

getModel(): KDMModel[0..1]	This operation returns the KDM model that owns the current KDM Entity.

10.3.4 Segment Class

The Segment element is a container for a meaningful set of facts about an existing software system. Each segment may include one or more KDM model instances and thus represents a collection of one or more *architectural views* for a given software system. Segment is a unit of exchange between the tools of the KDM ecosystem. Segment without owners is the top segment of the KDM model.

Superclass

KDMFramework

Associations

segment: Segment[0..*]	Nested Segment elements owned by the current Segment.
model[0..*]:KDMModel	The set of KDM models owned by the current segment. Each KDM model defines an architectural viewpoint. KDM model defines specific meta-model elements (entities and relationships specific to the viewpoint) that collectively define the viewpoint language.

Semantics

The implementer shall arrange KDM models into segments and to provide name attributes for each KDM segment instance. A KDM segment instance may be empty or may not contain KDM models of a given kind or may contain one or more KDM models of a given kind.

In general, KDM does not constrain associations between instances across KDM models or across KDM segments.

KDM meta-model patterns make it possible to arrange KDM instances in such a way that the models in KDM Program Elements and Infrastructure layers are placed in a separate KDM segment, which becomes a reusable asset for multiple derivative models from the Program Elements and Infrastructure layer.

The implementers of KDM import tools should not make any assumptions regarding the organization of the KDM models into KDM segments.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
        xmlns:source="http://schema.omg.org/spec/KDM/1.2/source" name="Framework Example">
  <audit xmi:id="id.0" description="Illustration of KDM Framework" author="KDM FTF" date="04-03-2007">
    <attribute xmi:id="id.1" tag="approved" value="yes"/>
  </audit>
  <segment xmi:id="id.2" name="foobar"/>
  <model xmi:id="id.3" xmi:type="code:CodeModel" name="foo">
    <annotation xmi:id="id.4" text="This is a sample instance of a Code model"/>
  </model>
```

```
  <model xmi:id="id.5" xmi:type="source:InventoryModel" name="bar">
    <annotation xmi:id="id.6" text="This is a sample of an Inventory model"/>
  </model>
</kdm:Segment>
```

10.4 Audit Class Diagram

The Audit class diagram defines meta-model elements to represent some extra “audit” information related to the KDM framework elements (models and segments). The classes and associations of the Audit class diagram are shown in Figure 10.2.

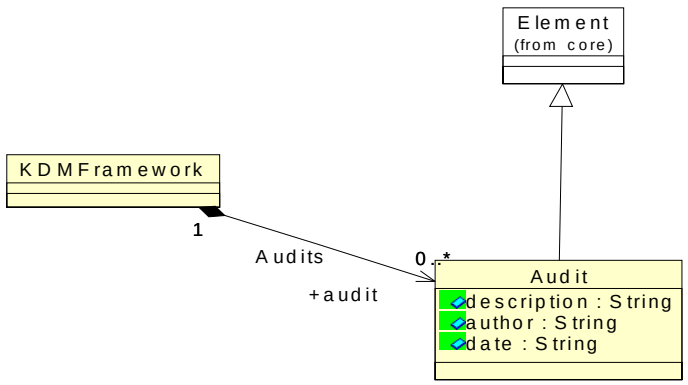

Figure 10.2 - Audit Class Diagram

10.4.1 Audit Class

Audit class represents basic audit information associated with KDM models.

Superclass

Element

Attributes

description:String	Contains the description of the Framework element.
author:String	Contains the name of the person who has created the model element, or the name of the tool that was used to create the model element.
date:String	Contains the date when the model element was created, in “dd-mm-yyyy” format.

Constraints

1. date should be represented in “dd-mm-yyyy” format

Semantics

The Audit element provides some extra “audit” information in the form of human readable text.

Each Framework element can have zero or more Audit instances associated with it. The collection of Audit elements is not ordered however the data property can be used to establish the temporal ordering. The Date format is “dd-mm-yyyy,” where “dd” is the date, the “mm” is the number of the month, with leading zero, if needed, and “yyyy” is the year. For example, “04-03-2007” corresponds to the “4th of March 2007.”

The content of the Audit element is provided by the implementer or the analyst.

KDM does not constrain the “description” attribute.

The implementers of KDM import tools should not make any assumptions regarding the content of the Audit element, other than the “human-readable text.” It is expected that at least one Audit element is related to the origin of the model or segment.

Audit element can own Annotation elements and Attribute elements. The Audit.description is the primary description and any associated annotations may be used as optional secondary descriptions.

Example

See example in the KDMFramework section.

10.4.2 KDMFramework (additional properties)

Associations

audit:Audit[0..*]	The list of Audit element instances for the given instance of KDMFramework (segment or model)

10.5 Extensions Class Diagram

The Extensions class diagram defines the meta-model elements that constitute the basis of the KDM light-weight extensions mechanism. Some additional meta-model elements are defined by the ExtendedValues class diagram.

The KDM light-weight extension mechanism is a standard way of adding new “virtual” meta-model elements to KDM. A “virtual” meta-model element is a base meta-model element with extended meaning, and possibly with extended attributes. The base meta-model element can be a “regular” element (a concrete class, not marked as “generic”), or “generic” (concrete class with under specified semantics, marked as “generic”). This mechanism is defined as part of KDM. The light-weight extensions mechanism allows introducing new “extended” meta-model element kinds (called stereotypes). The exact meaning and the intention of the “extended” meta-model elements is outside of KDM and should be communicated by implementers to the users of the extended representations.

The light-weight extension mechanism provides the following capabilities:

1. Define a stereotype (introduce the partial kind of a meta-model element):
    - A stereotype definition includes the name of class of the allowed base elements. This class can be a particular concrete meta-model element, a generic element, or an abstract meta-model element.
2. Define tags associated with a stereotype. Tags are additional attributes to the extended elements. Tag definition includes the name of the extended attribute and the name of the type of the element (represented as a string). Values of extended attributes can take the form of a string, or reference to some modeling element. Each stereotype defines its own set of tags. Tag definitions are owned by the corresponding stereotype definition.

3. Organize stereotype definitions into stereotype families. Stereotype families are owned by KDM Framework elements (KDM models and segments).

4. Use extended model elements in KDM instances by using the base meta-model element instance with one or more stereotype(s):

   - Concrete tag values can be added to the “virtual” element if the stereotype defines tags.
   - Each tag value is associated with the corresponding tag definition.
   - The complete kind of the new element is defined as the union of all stereotypes added to the element.

When added to a KDM model element, a stereotype provides additional semantic details to the base meta-model element. Stereotypes should not change the semantics of the base element.

KDM supports the light-weight extension mechanism through a Generic Element meta-model pattern. KDM defines a relatively large number of concrete meta-model elements with under specified semantic. In KDM these elements are the common superclasses for classes with specific semantics. However, generic elements can be used as *extension points* of the light-weight extension mechanism by using them in KDM instances with a stereotype. In addition, each KDM model defines two “wildcard” generic elements: a generic entity and a generic relationship for the given KDM model. They too can be used as extension points.

Pure KDM instances should use only concrete meta-model elements that have semantics specified in KDM, without stereotypes. KDM instances that use stereotypes are called extended KDM instances. Extended KDM instances can add stereotypes to concrete meta-model elements.

KDM light-weight extension mechanism does not support multiplicity of tags, constraints on tags, relationships between tags or stereotypes. The implementer shall select the most specific extension points, by defining stereotypes in such a way that they use the base elements with the most specific semantic description (closer to the bottom of the KDM class hierarchy).

The classes and associations of the Extensions diagram are shown in Figure 10.3.

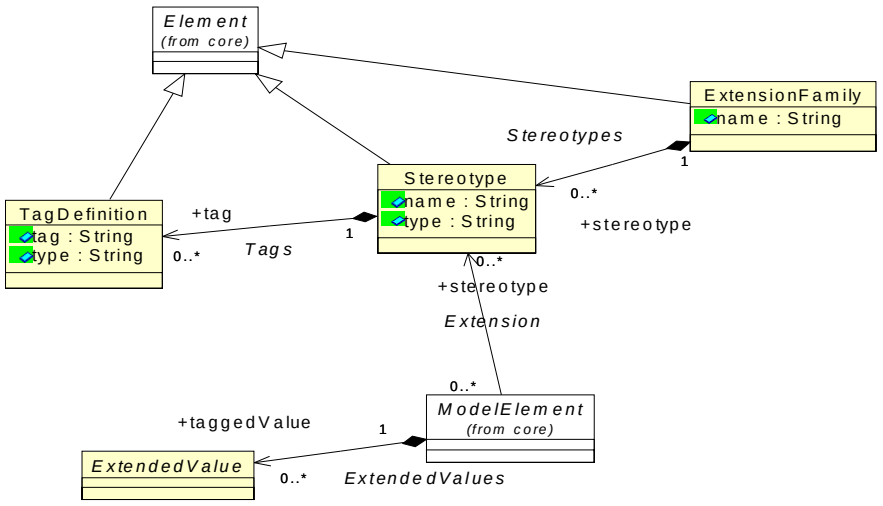

Figure 10.3 - Extensions Class Diagram

10.5.1 Stereotype Class

The stereotype concept provides a way of branding (classifying) model elements so that they behave as if they were the instances of new virtual meta-model constructs. These model elements have the same structure (attributes, associations, operations) as similar non-stereotyped model elements of the same kind. The stereotype may specify additional required tagged values that apply to model elements. In addition, a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure.

In the meta-model the Stereotype is a subclass of ModelElement. Stereotype is a named model element. TaggedValues attached to a Stereotype apply to all ModelElements branded by that Stereotype.

A Stereotype specifies the name of the base class to which it can be added.

Superclass

Element

Attributes

name:String	Specifies the name of the stereotype.
type:String	Specifies the name of the model element to which the stereotype applies.

Associations

tag:TagDefinition[0..*]	Stereotype owns the set of tag definitions that determine the additional tagged values associated with the model elements that are branded with the given stereotype.

Constraints

1. Tags associated with model element should not clash with any meta attributes associated with this model element.
2. A model element should have at most one tagged value with a given tag name.
3. A stereotype should not extend itself.
4. A Stereotype can be added to ModelElement if its class is the same as the baseClass, or one of its subclasses.
5. The values of the Type attribute of the TagDefinition are restricted to the names of the KDM meta-elements. Names of the core datatypes (“Boolean,” “String,” “Integer”) define attributes of the extended meta-model element. The corresponding values are represented as instances of the TaggedValue class. Names of other KDM meta-elements (for example, “KDMEntity,” or “Audit”) define associations of the extended meta-element and the corresponding values are represented as instances of the TaggedRef class.

Semantics

Stereotypes should not change the semantics of the base meta-model element.

A KDM model element with one or more stereotypes has the semantics of the base element with some additional semantic constraints. The complete kind of the new element is defined as the union of all stereotypes added to the element. Extended values are the attributes of the extended element. A KDM element with one or more stereotypes is equivalent to a situation in which KDM has been extended by adding a new meta-model class that extends the base class, with the new attributes corresponding to the tag definitions.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="Stereotype Example">
  <extensionFamily xmi:id="id.0" name="Example extensions">
    <stereotype xmi:id="id.1" name="Java method"/>
    <stereotype xmi:id="id.2" name="C++ method"/>
    <stereotype xmi:id="id.3" name="C++ procedure"/>
    <stereotype xmi:id="id.4" name="C++ friend">
      <tag xmi:id="id.5" tag="friend_of" type="ClassUnit"/>
    </stereotype>
    <stereotype xmi:id="id.6" name="IsFriendOf"/>
    <stereotype xmi:id="id.7" name="native call">
      <tag xmi:id="id.8" tag="implemented in" type="String"/>
    </stereotype>
  </extensionFamily>
  <model xmi:id="id.9" xmi:type="code:CodeModel" name="Example">
    <codeElement xmi:id="id.10" xmi:type="code:ClassUnit" name="myclass">
      <codeElement xmi:id="id.11" xmi:type="code:MethodUnit" stereotype="id.2"
                name="foo" type="id.12">
        <codeElement xmi:id="id.12" xmi:type="code:Signature" name="foo"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.13" xmi:type="code:CallableUnit" stereotype="id.4 id.3"
                name="bar" type="id.16" kind="regular">
      <taggedValue xmi:id="id.14" xmi:type="kdm:TaggedRef" tag="id.5" reference="id.10"/>
      <codeRelation xmi:id="id.15" xmi:type="code:CodeRelationship" stereotype="id.6"
                 to="id.10" from="id.13"/>
      <codeElement xmi:id="id.16" xmi:type="code:Signature" name="bar"/>
    </codeElement>
  </model>
  <model xmi:id="id.17" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.18" xmi:type="code:ClassUnit" stereotype="id.1">
      <codeElement xmi:id="id.19" xmi:type="code:MethodUnit" stereotype="id.1"
                name="foobar" type="id.23">
        <codeElement xmi:id="id.20" xmi:type="action:ActionElement" stereotype="id.7"
                  name="a1">
          <actionRelation xmi:id="id.21" xmi:type="action:Calls" stereotype="id.7"
                     to="id.13" from="id.20">
            <taggedValue xmi:id="id.22" xmi:type="kdm:TaggedValue" tag="id.8" value="C"/>
          </actionRelation>
        </codeElement>
        <codeElement xmi:id="id.23" xmi:type="code:Signature" name="foobar"/>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

10.5.2 TagDefinition Class

Lightweight extensions allows information to be attached to any model element in the form of a “tagged value” pair (i.e., name=value). The interpretation of tagged value semantics is outside the scope of KDM. It must be determined by the user or tool conventions. It is expected that tools will define tags to supply information needed for their operations beyond the basic semantics of KDM. Such information could include specific entities, relationships, and attributes for a particular programming language, runtime platform, or engineering environment.

Even though TaggedValues is a simple and straightforward extension technique, their use restricts semantic interchange of extended information about existing software systems to only those tools that share a common understanding of the specific tagged value names.

Each Stereotype owns the optional set of TagDefinitions. Each TagDefinition provides the name of the tag and the name of the KDM type of the corresponding value.

In the meta-model, TagDefinition is a subclass of Element.

Superclass

Element

Attributes

tag:String	Contains the name of the tagged value. This name determines the semantics that are applicable to the contents of the value attribute.
type:String	Specifies the type of the value attribute.

Constraints

1. The “value” attribute of the TaggedValue should be valid according to the type specified in the corresponding TagDefinition.
2. The target of the “ref” association of the TaggedRef should be of the type specified in the corresponding TagDefinition, or one of its subtypes.
3. If the type of the TaggedDefinition is one of the primitive datatypes (for example, “BooleanType,” “StringType,” “IntegerType”), the corresponding value should be an instance of the TaggedValue class.
4. If the type of the TaggedDefinition is a name of some other KDM meta-element (for example, “KDMEntity,” or “Audit”), the corresponding value should be an instance of the TaggedRef class.

Semantics

ExtendedValues provide the values of the extended attributes and associations of the extended meta-model element defined by one or more stereotypes.

Names of the tags, defined by each stereotype, constitute an isolated namespace that does not interfere with the names of other stereotypes or the names of the attributes of the base type. The meaning and the intention of the stereotypes and tags is outside of the KDM specification and should be communicated by implementers to the users of the extended models. Extensions should not change the semantics of the base KDM meta-model elements, so that the model with extensions can still be interpreted as an approximation of the full extended meaning when extensions are ignored and only the basic KDM semantic rules are applied.

Example

See example in the Stereotype class section.

10.5.3 ExtensionFamily Class

ExtensionFamily provides a mechanism for managing lightweight extensions. ExtensionFamily acts as a container for a set of related stereotypes and their corresponding tag definitions.

Superclass

Element

Attributes

name:String	Provides the name of the extension family.

Associations

stereotype:Stereotype[0..*]	The set of stereotypes that are owned by the extension family.

Semantics

ExtensionFamily provides a named container for stereotype definitions. The implementer shall arrange stereotype definitions into meaningful families. Some stereotype definitions may be made available as reusable assets in a separate segment. KDM analysis tools should not make any assumptions regarding the organization of stereotypes into families.

Example

See example in the Stereotype class section.

10.5.4 ModelElement (additional properties)

Associations

taggedValue:TaggedValue[0..*]	The set of tagged values determined by the stereotype.
stereotype:Stereotype[0..*]	The stereotype

Constraints

1. Each tagged value added to a ModelElement must conform to a certain tag definition owned by the stereotype of that ModelElement (the tag association of the TaggedValue should refer to a TaggedDefinition that is owned by a Stereotype of the ModelElement). A tagged value conforms to a tag definition when the value conforms to the type of the TagDefinition. Conformance of lightweight extensions can only be validated dynamically by a suitable KDM import tool, since lightweight extensions are not defined by the KDM specification.
2. Stereotype can be associated with a certain instance of a ModelElement if the type of the ModelElement is the same as the type property in the stereotype definition, or one of its subclasses.

Example

See example in the Stereotype class section.

10.6 ExtendedValues Class Diagram

ExtendedValues class diagram defines additional meta-model constructs as part of the KDM light-weight extension mechanism. These constructs represent extended values that can be added to extended KDM model elements. The key of the light-weight extension mechanism is a meta-model construct called stereotype. Stereotypes provide additional meaning to KDM meta-model constructs. While the meaning of a stereotype is not defined within the KDM, stereotypes allow differentiation within existing KDM metamodel elements and allow adding attributes to them with extended value construct.

The classes and associations involved in the definition of extended values are shown in Figure 10.4.

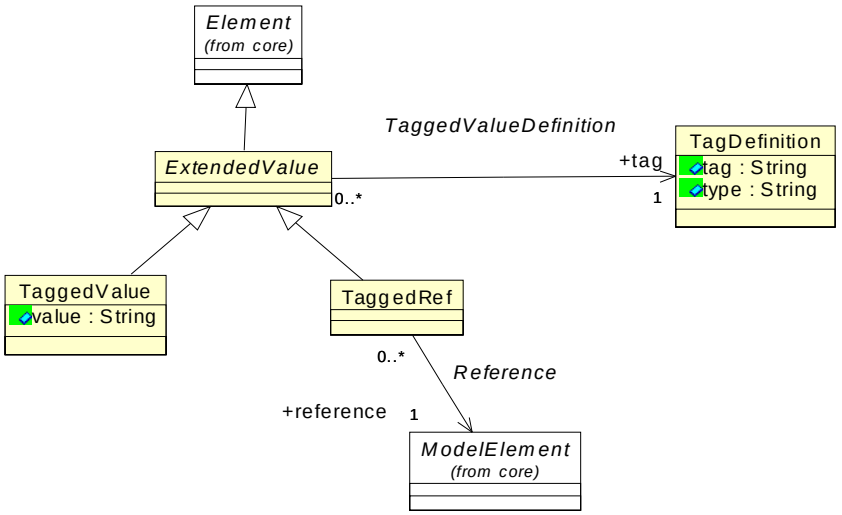

Figure 10.4 - ExtendedValue Class Diagram

10.6.1 ExtendedValue Class (abstract)

ExtendedValue class is an abstract superclass for the two concrete classes that represent tagged values: the TaggedValue and the TaggedRef. ExtendedValue class defines common properties for these classes.

Superclass

Element

Associations

tag [1]:TagDefinition	The reference to the tag definition of the corresponding stereotype.

Semantics

ExtendedValue is a “virtual” attribute to an extended KDM meta-model element. ExtendedValue element represents the value of the attribute. KDM defines two concrete subclasses of ExtendedValue: TaggedValue and TaggedRef. The definition of the attribute is provided by the TagDefinition element owned by a Stereotype element. The Stereotype element defines the “virtual” meta-model element that provides the context for the new attributes. “Virtual” attributes are instantiated every time a new “virtual” meta-model element, defined by a Stereotype is instantiated. This is an important difference between ExtendedValues and KDM attributes, which are not related to any particular meta-model element.

The TagDefinition element defines constraint on the possible values of the ExtendedValue by specifying the type of the allowed values.

Each instance of ExtendedValue has an association to the corresponding TagDefinition.

10.6.2 TaggedValue Class

A tagged value allows information to be attached to any model element in the form of a “tagged value” pair, (i.e., name=value). The interpretation of tagged value semantics is outside of the scope of KDM. It must be determined by the user or tool conventions. It is expected that tools will define tags to supply information needed for their operations beyond the basic semantics of KDM. Such information could include specific entities, relationships, and attributes for a particular programming language, runtime platform, or engineering environment.

Each TaggedValue must conform to the corresponding TagDefinition. In the meta-model, TaggedValue is a subclass of Element.

Superclass

ExtendedValue

Attributes

Value:String	Contains the current value of the TaggedValue.

Constraints

1. The value of the TaggedValue instance should conform to the type of the corresponding TagDefinition.

Semantics

TaggedValue element represents simple atomic “virtual” attributes. The type constraint of the value, defined in the corresponding TagDefinition can be the name of any KDM primitive type (for example, “StringType,” “BooleanType,” etc.).

Example

See example in the Stereotype class section.

10.6.3 TaggedRef Class

A TaggedRef allows information to be attached to any model element in the form of a reference to another exiting model element. Each TaggedRef must conform to the corresponding TagDefinition: the actual type of the model element that is the target of the TaggedRef must be the same as the type specified in the corresponding TagDefinition, or one of its subtypes. In the meta-model, TaggedRef is a subclass of ExtendedValue.

Superclass

ExtendedValue

Associations

ref:ModelElement[1]	Designates the model element referred to by the extended value.

Constraints

1. The model element that is the target of the ref association must be of the type, specified by the type attribute of the tag definition that is the target of the tag association of the tagged ref element.

Semantics

TagRef represents complex “virtual” attributes, which are associations to other KDM elements. TagDefinition can be a name of any KDM meta-model element (for example, “KDMEntity,” “AbstractCodeElement,” “ControlElement,” or “CallableUnit”).

Example

See example in the Stereotype class section.

10.7 Annotations Class Diagram

The Annotation class diagram defines meta-model elements that allow ad hoc user-defined attributes and annotations to instances of KDM elements. The mechanism of ad hoc user-defined attributes provides a capability to add pairs <tag, value> to an individual element instance. This is complimentary to the light-weight extension mechanism, which provides a new meta-model element, each instance of which has <tag, value> pair specified by the definition of the extended element. An ad hoc user-defined attribute is owned to an individual element instance. This means that different instances of the same meta-model element may own completely different user-defined attributes (and some may have none at all).

An Annotation is an ad hoc note owned by an individual element instance. Annotations and attributes are applied to the elements of KDM instances. They may be used by implementer to add specific information to an individual element. They may also be used by an analyst, annotating a given KDM instance. On the other hand, stereotypes and tag definitions as first defined as extensions to the KDM (meta-model) and then systematically used by the implementer.

The classes and associations that make up the Annotations diagram are shown in Figure 10.5.

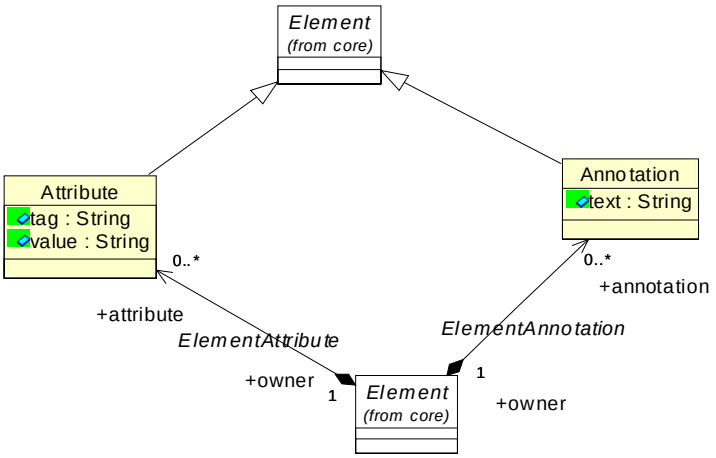

Figure 10.5 - Annotations Class Diagram

10.7.1 Attribute Class

An attribute allows information to be attached to any model element in the form of a “tagged value” pair (i.e., name=value). Attribute add information to the instances of model elements, as opposed to stereotypes and tagged values, which apply to meta-model elements. Tagged value is part of the extension mechanism (stereotypes define virtual new model element, and tagged values specify additional attributes of these virtual model elements). Tagged values are only associated with model elements branded by a stereotype, and the set of tagged values for a particular instance of a model element is determined by its stereotype. On the other hand, arbitrary attributes may be associated with individual instances of model element. In particular, two different instances of the same model element may be annotated by different attributes.

In the meta-model, TaggedValue is a subclass of Element.

Superclass

Element

Attributes

tag:Name	Contains the name of the attribute. This name determines the semantics that are applicable to the contents of the value attribute.
value:String	Contains the current value of the attribute.

Constraints

1. Attribute cannot have further annotations or attributes

Semantics

The interpretation of attribute semantics is outside the scope of KDM. It may be determined by the user or the implementer conventions. Tools may provide capability to add arbitrary attributes to the instances of the model to supply information needed for their operations beyond the basic semantics of KDM. Such information can support analysis of KDM models by analysis, etc.

An attribute element is not related to a particular meta-model element. It does not define a “virtual” attribute to an extended meta-model element, that is instantiated with every instantiation of the new element. Instead, an attribute element can be added to any KDM element. It defines a property of a particular instance, not a property of a class of instances.

Example

See example in the KDMFramework section.

10.7.2 Annotation Class

Annotations allow textual descriptions to be attached to any instance of a model element. The meta-model Annotation class is a subclass of Element.

Superclass

Element

Attributes

text:String	Contains the text of the annotation to the target model element.

Constraints

1. Annotation cannot have further annotations or attributes.

Semantics

Annotation allows associating a human-readable text with an instance of any Element.

Example

See example in the KDMFramework section.

10.7.3 Element (additional properties)

Associations

attribute:Attribute[0..*]	The set of attributes owned by the given element.
annotation:Annotation[0..*]	The set of annotations owned by the given element.

Semantics

No assumptions should be made regarding the order of attributes or annotations associated with a particular instance.

11 Source Package

11.1 Overview

The Source package defines a set of meta-model elements whose purpose is to represent the physical artifacts of the existing system, such as source files, images, configuration files, resource descriptions, etc. The Source package also represents traceability links between instances of KDM meta-elements and the regions of source code, which is represented by these meta-model elements. It represents the link between the KDM instance and the artifacts of the existing system it represents.

The Source package offers two capabilities for linking instances of the KDM representation to the corresponding artifacts:

- Inlining the corresponding source code in the form of a “snippet” into KDM representation
- Linking a KDM element to a region of the source code within some physical artifact of the system being modeled

A given KDM representation can implement either of the approaches, both of them, or none.

When a KDM element is linked to the source code within a particular physical artifact of the existing system (regardless of the existence of the corresponding snippet), KDM offers further two options:

- The link can utilize the element of the KDM inventory model to identify the particular physical artifact, in which case the path to the artifact is determined through the Inventory Model.
- The link can be made stand-alone and explicitly specify the path to the artifact.

The nature of the “source code” represented by a particular KDM element is unspecified within KDM. In KDM, this is indicated by the “language” attribute.

The Source package defines an *architecural viewpoint* for the Inventory domain. It is determined by the entire software development environment of the existing software system.

- **Concerns**
  - What are the artifacts (software items) of the system?
  - What is the general role of each artifact (for example, is it a source file, a binary file, an executable, or a configuration description)?
  - What is the organization of the artifacts (into directories and projects)?
  - What are the dependencies between the artifacts?
- **Viewpoint language**

  Inventory views conform to KDM XMI schema. The viewpoint language for the Inventory architectural viewpoint is defined by the Source package. It includes an abstract entity AbstractInventoryElement, several generic entities, such as InventoryItem and InventoryContainer, as well as several concrete entities, such as SourceFile, BinaryFile, Image, Directory, etc. The viewpoint language for the Inventory architectural viewpoint also includes DependsOn relationship, which are subclasses of AbstractInventoryRelationship.
- **Analytic methods**

  The Inventory architectural viewpoint supports the following main kinds of checking:

    - What artifacts depend on the given artifact?
    - The Inventory viewpoint also supports check in combinations with other KDM architectural viewpoint to determine the original artifacts that correspond to a given KDM element.
- **Construction methods**
    - Inventory views that correspond to the KDM Inventory architectural viewpoint are usually constructed by directory scanning tools, which identify files and their types.
    - Construction of an Inventory view is determined by the particular development and deployment environments of the existing software system.
    - Construction of an Inventory view is determined by the semantics of the environment as well as the semantics of the corresponding artifacts, and it based on the mapping from the given environment to KDM.
    - The mapping from a particular environment to KDM may produce additional information (system-specific, or environment-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes, or annotations.

As a general rule, in a given KDM instance, each instance of the inventory model represents a file, or a set of files. Exceptions to this rule are:

- InventoryModel element, which is a part of the KDM instance infrastructure. This meta-model element is a container the instances of other inventory meta-model elements.
- SourceRef and SourceRegion meta-elements that represent traceability links between other instances of KDM meta-model elements and source code of the existing software system.

Inventory meta-model elements are part of the KDM instance infrastructure because they determine the traceability mechanism between other KDM elements and the regions of the physical artifacts of the existing software system that they represent.

11.2 Organization of the Source Package

The Source package consists of the following 5 class diagrams:

- InventoryModel
- InventoryInheritances
- InventoryRelations
- SourceRef
- ExtendedInventoryElements

The Source package depends on the following packages:

- Core
- kdm

11.3 InventoryModel Class Diagram

InventoryModel class diagram defines meta-model elements that represent the physical artifacts of the existing software system. This model corresponds to the *inventory* of the artifacts. The InventoryModel class diagram follows the uniform pattern for KDM model to extend the KDM Framework with specific meta-model elements. InventoryModel defines the following meta-model elements determined by the KDM model pattern:

- InventoryModel class
- AbstractInventoryElement class
- AbstractInventoryRelationship class

In addition, the InventoryModel class diagram defines a concrete KDM entity for each artifact, such a SourceFile, an Image, a ResourceDescription, a Configuration description, a BinaryFile, and an ExecutableFile. These meta-model elements are subclasses of the common parent class InventoryItem. The Inventory model also provides a generic KDM container called InventoryContainer and two specific containers: Directory and Project.

The classes and associations of the InventoryModel are shown at Figure 11.1.

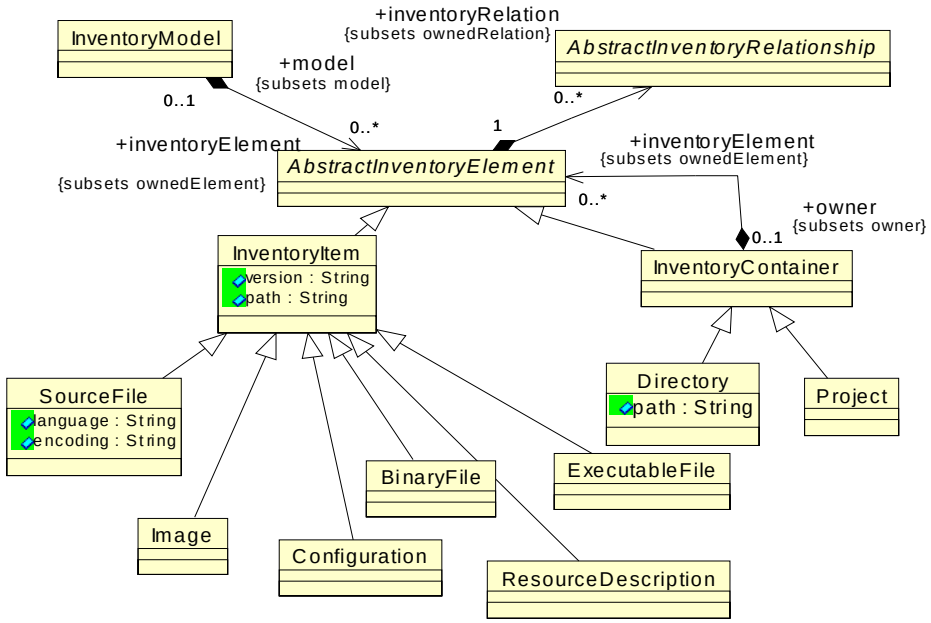

Figure 11.1 - InventoryModel Class Diagram

11.3.1 InventoryModel Class

The InventoryModel is a specific KDM model that owns collections of facts related to the physical artifacts of the existing software system. InventoryModel is a container for the instances of InventoryItems. InventoryModel corresponds to the *inventory* of the physical artifacts of the existing software system.

Superclass

KDMModel

Associations

inventoryElement:AbstractInventoryElement[0..*]	The set of inventory elements owned by the inventory model.

Semantics

InventoryModel is a container for instances of inventory elements. The implementer shall arrange inventory elements into one or more inventory models. KDM import tools shall not make any assumptions about the organization of inventory items into inventory models.

11.3.2 AbstractInventoryElement Class (abstract)

The AbstractInventoryElement is the abstract parent class for all inventory entities.

Superclass

KDMEntity

Associations

inventoryRelationship:AbstractInventoryRelationship[0..*]	The set of inventory relations owned by the inventory element.

Semantics

From the meta-model perspective, this element is a common parent for all inventory entities. This element is abstract and cannot occur in KDM instances. The name of the meta-model element can be used as the type constraint in stereotype definitions. Inventory package provides a concrete class InventoryElement with under specified semantics, which can be used as an extension point for defining new “virtual” inventory entities.

11.3.3 AbstractInventoryRelationship Class (abstract)

The AbstractInventoryRelationship is the abstract parent class for all inventory relationships.

Superclass

KDMRelationship

Constraints

Semantics

From the meta-model perspective, this element is a common parent for all inventory relationships. This element is abstract and cannot occur in KDM instances. The name of the meta-model element can be used as the type constraint in stereotype definitions. Inventory package provides a concrete class InventoryRelationship with under specified semantics, which can be used as an extension point for defining new “virtual” inventory relationships.

11.3.4 InventoryItem Class (generic)

InventoryItem is a generic meta-model element that represents any artifact of an existing software system. This class is further subclasses by several concrete meta-model elements with more precise semantics. However, InventoryItem can be used as an extended modeling element with a stereotype.

Superclass

AbstractInventoryElement

Attributes

version:String	Provides the ability to track version or revision numbers.
path:String	Location of the build resource.

Semantics

The implementer shall provide a mapping from concrete types of the physical artifacts involved in the engineering of the existing software system into concrete subclasses of the InventoryItem. The implementer shall map each artifact of the existing software system to some instance of KDM InventoryItem.

11.3.5 SourceFile Class

The SourceFile class represents source files. This meta-model element is the key part of the traceability mechanism of KDM whose purpose is to provide links between code elements and their physical implementations using the SourceRegion mechanism from the Source package.

Instances of the SourceRegion meta-model element refer to certain regions of source files to identify the original representation corresponding to a certain KDM element. In order to achieve interoperability between KDM tools that take advantage of the traceability links between the KDM elements and the regions of existing software system artifacts, KDM specification explicitly defines semantics of source files.

Superclass

InventoryItem

Attributes

language:String	Indicates the language of the source file.
encoding:String	An optional attribute that represents the encoding of the characters in the file.

Semantics

KDM assumes that a source file is a sequence of lines, identified by a linenumber. Each line is a sequence of characters, identified by a position within the line. Whitespace characters like tabulation are considered to be a single character. The “end of line” character is not considered to be part of the line.

KDM tools may use the information from the artifacts of the existing software system, accessible through the SourceRegion mechanism. It is recognized that different encodings are used around the world, and it may be desired for KDM processors to read code snippets from the files that use them. The requirement for KDM tools is to read information in UTF-8.

Specification of encoding is aligned with the XML specification from W3C. Each artifact of an existing system may use a different encoding for its characters. The default encoding for SourceFile is "UTF-8." Encodings other that UTF-8 should be explicitly specified in the optional encoding attribute of the SourceFile using a standard encoding label. For example, "UTF-16," "ISO-10646-UCS-2," "ISO-8859-2," "ISO-2022-JP," "Shift_JIS," and "EUC-JP," etc. Encoding of the characters in the SourceFile should be taken into account by KDM consumer tools that make use of the information in the source file, through the mechanism of the SourceRegion.

11.3.6 Image Class

The Image is used to represent image files.

Superclass

InventoryItem

Semantics

11.3.7 Configuration Class

The Configuration is used to represent various configuration files.

Superclass

InventoryItem

Semantics

11.3.8 ResourceDescription Class

The ResourceDescription is used to represent resource description files.

Superclass

InventoryItem

Semantics

11.3.9 BinaryFile Class

The BinaryFile is used to represent binary files.

Superclass

InventoryItem

Semantics

11.3.10 ExecutableFile Class

The ExecutableFile is used to represent executable files for a particular platform.

Superclass

InventoryItem

Semantics

11.3.11 InventoryContainer Class (generic)

The InventoryContainer meta-model element provides a container for instances of InventoryItem elements.

Superclass

AbstractInventoryElement

Associations

inventoryElement:AbstractInventoryElement[0..*]	The set of inventory elements owned by the container.

Constraints

1. InventoryContainer should have at least one stereotype.

Semantics

Concrete instances of the InventoryContainer element own other inventory elements (both inventory containers and individual inventory items). InventoryContainer instances represent tree-like container structures in which the leaf elements are individual InventoryItem instances. Each InventoryContainer represents the entity set of InventoryItems owned by that container directly or indirectly.

11.3.12 Directory Class

The Directory class represents directories as containers that own inventory items.

Superclass

InventoryContainer

Attributes

path:String	Location of the directory

Semantics

Directory items represent physical containers for the artifacts of the existing software systems, for example directories in file systems.

In addition to the general semantics of the InventoryContainer, Directory ownership structure determines the full “path” for each individual inventory item in the following way. For a given Directory item, the full “path” to an inventory item, owned by this Directory directly or indirectly, is a sequence of strings, the first element of which is the “path” attribute of

the Directory, and subsequent elements are name attributes of the directory items such that each directory item is owned by the previous directory item and that last directory item owns the inventory item. Any Project containers, involved in this ownership structure are ignored.

11.3.13 Project Class

The Project meta-model element represents an arbitrary logical container for inventory items.

Superclass

InventoryContainer

Semantics

Project is an arbitrary container for Inventory items. It can be used in combination with Directory containers.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
       xmlns:source="http://schema.omg.org/spec/KDM/1.2/source" name="Inventory Example">
  <model xmi:id="id.0" xmi:type="source:InventoryModel">
    <inventoryElement xmi:id="id.1" xmi:type="source:SourceFile" name="a.c">
      <inventoryRelation xmi:id="id.2" xmi:type="source:DependsOn" to="id.5" from="id.1"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.3" xmi:type="source:SourceFile" name="b.c">
      <inventoryRelation xmi:id="id.4" xmi:type="source:DependsOn" to="id.5" from="id.3"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.5" xmi:type="source:SourceFile" name="ab.h"/>
    <inventoryElement xmi:id="id.6" xmi:type="source:Directory">
      <inventoryElement xmi:id="id.7" xmi:type="source:Image"/>
      <inventoryElement xmi:id="id.8" xmi:type="source:Image"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.9" xmi:type="source:SourceFile" name="makefile"/>
    <inventoryElement xmi:id="id.10" xmi:type="source:ExecutableFile" name="ab.exe"/>
  </model>
</kdm:Segment>
```

11.4 InventoryInheritances Class Diagram

InventoryInheritances class diagram is determined by the KDM model pattern. This diagram defines how the classes of the InventoryModel extend the KDM Framework. The classes and associations for this diagram are shown in Figure 11.2.

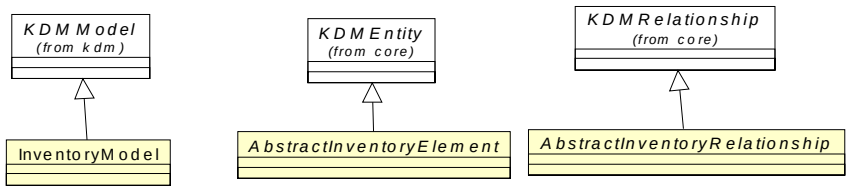

Figure 11.2 - InventoryInheritances Class Diagram

11.5 InventoryRelations Class Diagram

InventoryRelations class diagram defines an optional relationship “DependsOn” between inventory elements. The classes and associations for this diagram are shown in Figure 11.3.

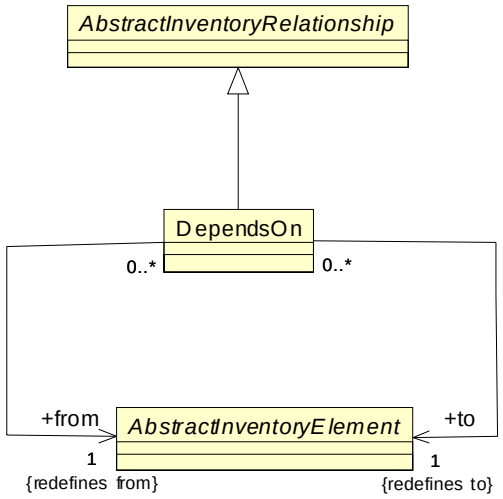

Figure 11.3 - InventoryRelations Class Diagram

11.5.1 DependsOn Class

DependsOn class is a meta-model element that represents an optional relationship between two inventory items, in which one inventory element requires another inventory element during one or more steps of the engineering process.

Associations

from:AbstractInventoryElement[1]	The base inventory item.
to:AbstractInventoryElement[1]	Another inventory item on which the base item depends.

Constraints

1. An inventory item should not depend on itself.

Semantics

The DependsOn relationship is optional. The implementer may capture certain aspects - knowledge of the engineering process in the form of “DependsOn” relations between inventory items. “DependsOn” relationship is part of the Infrastructure Layer, which is available to all KDM implementations at various compliance levels. KDM Build package that constitutes a separate L1.Build compliance point, defines additional meta-model elements that represent the facts involved in the build process of the software system (including but not limited to the engineering transformations of the “source code” to “executables”).

When the origin of the DependsOn relationship is an Inventory container, this means that all elements owned by this container (directly or indirectly) depend on the target of the relationship.

When the target of the “DependsOn” relationship is an Inventory container, this means that the one or more base inventory elements (the origin of the relationship) depends on all elements owned by the container (directly or indirectly).

11.6 SourceRef Class Diagram

The SourceRef class diagram defines a set of meta-model elements whose purpose is to provide traceability links between the elements of the KDM model of the existing software system and the physical artifacts of that system. The class diagram shown in Figure 11.4 captures these classes and their relations.

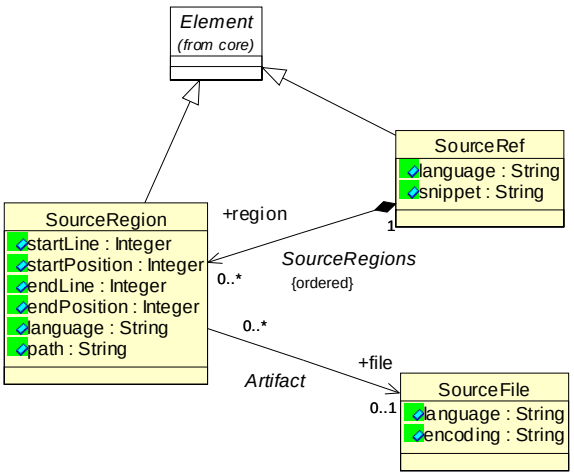

Figure 11.4 - SourceRef Class Diagram

11.6.1 SourceRef Class

The SourceRef class represents a traceability link between a particular model element and the corresponding source code.

Superclass

Element

Attributes

language: String	Optional attribute - indicates the source language of the snippet attribute.
snippet:String	Optional attribute - the source snippet for the given KDM element. The snippet may have some internal structure, for example XML tags corresponding to an abstract syntax tree of the code fragment. The interpretation of code snippets is outside the scope of the KDM.

Constraints

1. Language indicator has to be provided using at least one of the following methods:
   - As the attribute of the SourceRef element.
   - As the attribute of the SourceRegion element.

• As the attribute of the SourceFile element (part of the Inventory Model), accessible via the SourceRegion element.

2. If both the snippet and the language attributes of the SourceRef element are present, then the language attribute should describe the nature of the code snippet, in which case the nature of the source code region accessible through the SourceRegion may be different from the nature of the code snippet. If the snippet attribute is not present, then the language attribute of the SourceRef element overrides the language attribute of the SourceRegion element, which in turn overrides the one at the SourceFile element.

Semantics

SourceRef meta-model element represents a traceability link between an instance of a KDM element to its original "source" representation as part of a physical artifact of the existing software system. KDM element that defines a traceability link to its original representation owns one or more SourceRef elements.

The Source package offers two capabilities for linking instances in KDM representation to their corresponding artifacts:

- Inlining the corresponding source code in the form of a "snippet" into KDM representation.
- Linking a KDM element to a region of the source code within some physical artifact of the system being modeled.

A given KDM representation can implement either of the approaches, both of them, or none.

When a KDM element is linked to the source code within a particular physical artifact of the existing system (regardless of the existence of the corresponding snippet), KDM offers further two options:

- The link can utilize the element of the KDM inventory model to identify the particular physical artifact, in which case the path to the artifact is determined through the Inventory Model.
- The link can be made stand-alone and explicitly specify the path to the artifact.

KDM element can define more than one SourceRef traceability link. The first SourceRef element is considered as the primary one and other elements are considered secondary. Secondary traceability links may be used to represent alternative views of the code, links to other artifacts (such as design and documentation), represent generated code or target code during the software modernization process. Usually, secondary SourceRef elements have distinct "language" attributes, so that KDM tools can select the appropriate representation to display.

The implementer shall provide adequate traceability links.

11.6.2 SourceRegion Class

The SourceRegion class provides a pointer to a single region of source. The SourceRegion element provides the capability to map precisely model elements to a particular region of source that is not necessarily text. The nature of the source code within the physical artifact is indicated by the language attribute of the SourceRegion element or the language attribute of the SourceFile element. The language attribute of the SourceRegion element overrides that of the SourceFile element if both are present.

The source region is located within some physical artifact of the existing software system (a source file).

Superclass

Element

Attributes

startLine: Integer	The line number of the first character of the source region.
startPosition:Integer	Provides the position of the first character of the source region.
endLine:Integer	The line number of the last character of the source region.
endPosition:Integer	The position of the last character of the source region.
language:String	Optional attribute - the language indicator of the source code for the given source region.
path:String	Optional attribute - the location of the physical artifact that contains the given source region.

Associations

file:SourceFile[0..1]	This association allows zero or more SourceRegion elements to be associated with a single SourceFile element of the Inventory Model.

Constraints

1. The location of the source file should be provided using at least one of the following methods:
   - Path attribute of the SourceRegion element.
   - Path attribute of the SourceFile element of the Inventory model.

Semantics

KDM assumes that a source file is a sequence of lines, identified by a linenumber. Each line is a sequence of characters, identified by a position within the line. Whitespace characters like tabulation are considered to be a single character. The “end of line” character is not considered to be part of the line.

The path attribute should uniquely identify the physical artifact. The nature of the path attribute is outside of the scope of the KDM. For example, this can be a URI.

Individual SourceRef elements may own multiple SourceRegion elements that represent a situation where there are multiple disjoint regions of source code that correspond to the given KDM element.

11.7 ExtendedInventoryElements Class Diagram

The ExtendedInventoryElements class diagram defines two “wildcard” generic elements for the inventory model as determined by the KDM model pattern: a generic inventory entity and a generic inventory relationship. The classes and associations of the ExtendedInventoryElements diagram are shown in Figure 11.5.

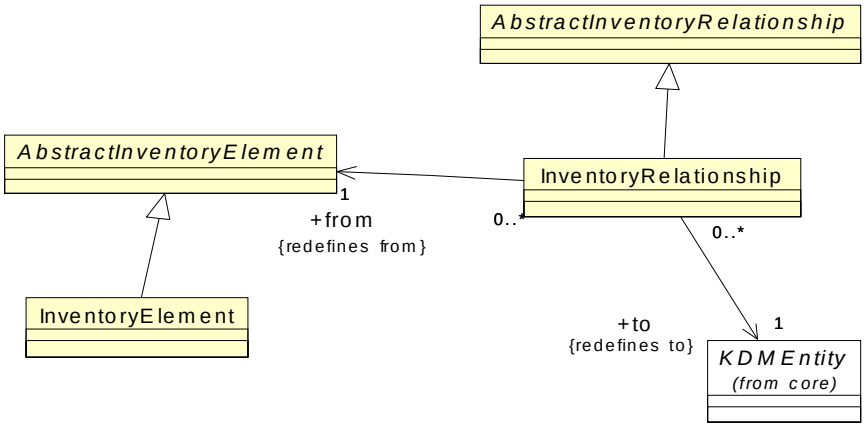

Figure 11.5 - ExtendedInventoryElements Class Diagram

11.7.1 InventoryElement Class (generic)

The InventoryElement class is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractInventoryElement

Constraints

1. InventoryElement should have at least one stereotype.

Semantics

An inventory entity with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model entity type of the inventory model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

11.7.2 InventoryRelationship Class (generic)

The InventoryRelationship class is a generic meta-model element that can be used to define new "virtual" inventory relationships through the KDM light-weight extension mechanism.

Superclass

AbstractInventoryRelationship

Associations

from:AbstractInventoryElement[1]	The inventory element origin endpoint of the relationship.
to:KDMEntity[1]	The target of the relationship.

Constraints

1. InventoryRelationship should have at least one stereotype.

Semantics

An inventory relationship with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model relationship type of the inventory model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

Part II - Program Elements Layer

The Program Elements Layer defines a large set of meta-model elements whose purpose is to provide a language-independent intermediate representation for various constructs determined by common programming languages.

Packages of the Program Elements Layer define an *architecture viewpoint* for the Code domain.

- **Concerns**
  - What are the computational elements of the system?
  - What are the modules of the system?
  - What is the low-level organization of the computational elements?
  - What are the datatypes used by the computational elements?
  - What are the units of behavior of the system?
  - What are the low-level relationships between the code elements, in particular what are the control-flow and data-flow relationships?
  - What are the important non-computational elements?
  - How are computational elements and modules related to the physical artifacts of the system?
- **Viewpoint language**

  Code views conform to KDM XMI schema. The *viewpoint language* for the Code *architectural viewpoint* is defined by the Code and Action packages. It includes several abstract entities, such as AbstractCodeElement and CodeItem, several generic entities, such as Datatype, ComputationalObject and Module, as well as several concrete entities, such as StorableUnit, CallableUnit, CompilationUnit, and ActionElement. The viewpoint language for the Code architectural viewpoint also includes several relationships, which are subclasses of AbstractCodeRelationship and AbstractActionRelationship.
- **Analytic methods**

  The Code architectural viewpoint supports the following main kinds of checking:
  - Composition - for example, what code elements are owned by a CompilationUnit, SharedUnit, or a CodeAssembly; what action elements are owned by a CallableUnit.
  - Data flow - for example, what action elements read from a given StorableUnit; what action elements write to a given StorableUnit; what action elements create dynamic instances of a given Datatype; what action elements address a particular StorableUnit; what data element are being read as actual parameters in a call.
  - Control flow - for example, what CallableUnit is used in a call; what action element is executed after the given action element; what action elements are executed before the given action element; what data element is used to dispatch control flow from a given action element; what action element is executed after the given element under what conditions; what is the exceptional flow of control; what action elements are executed as entry points to a given module or a CallableUnit.
  - Datatypes - for example, what is the datatype of the given storable unit; what is the base datatype of the given pointer type; what is the base datatype of the given element of the record type; what is the signature of the given CallableUnit.

Other kinds of checking are related to the interfaces, templates, and pre-processor. All relationships defined in the Code model are non-transitive. Additional computations are required to derive, for example, all action elements that can be executed after the given action element, or all CallableUnits that a given action element can dispatch control to.

The KDM mechanism of aggregated relationship is used to derive relationships between KDM elements that own or reference various Code elements (usually Module and CodeAssembly) based on the low-level relationship between individual Code elements.

- **Construction methods**
  - Code views that correspond to the KDM Code architectural viewpoint are usually constructed by parser-like tools that take artifacts of the system as the input and produce one or more Code views as output.
  - Construction of the Code view is determined by the syntax and semantics of the programming language of the corresponding artifact, and it based on the mapping from the given programming language to KDM; such mapping is specific only to the programming language and not to a specific software system.
  - The mapping from a particular programming language to KDM may produce additional information (system-specific, or programming language-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes, or annotations.

Program Layer defines a single KDM Model, called CodeModel, and consists of the following KDM packages:

- Code
- Action

Code package defines CodeItems (named elements determined by the programming language, the so-called “symbols,” “definitions,” etc.) and *structural* relations between them. CodeItems are further categorized into ComputationalObject, Datatypes, and Modules. Action package defines behavioral elements and various behavioral relationships, which detemine the control- and data- flows between code items.

Description of the Code package is further subdivided into the following parts:

- Code Elements representing Modules
- Code Elements representing Computational Objects
- Code Elements representing Datatypes
- Code Elements representing Preprocessor Directives
- Miscellaneous Code Elements

Data representation of KDM is aligned with ISO/IEC 11404 (General-Purpose Datatypes) standard. In particular, KDM provides distinct meta-model elements for “data elements” (for example, global and local variables, constants, record fields, parameters, class members, array items, and pointer base elements) and “datatypes.” Each data element has an association “type” to its datatype. KDM distinguishes primitive datatypes (for example Integer, Boolean), complex user-defined datatypes (for example, array, pointer, sequence) and named datatypes (for example, a class, a synonym type). KDM meta-model elements corresponding to datatypes are subclasses of a generic class Datatype. KDM meta-model elements corresponding to data elements are subclasses of a generic class DataElement.

KDM model elements represent existing artifacts determined by a programming language. KDM meta-model elements provide sufficient coverage for most common datatypes and data elements, common to programming languages. KDM also provides several powerful generic extensible elements that can be further used with stereotypes to represent uncommon situations.

In addition to the type association, KDM relationship “HasType” is used to track named datatypes. Anonymous datatypes can be owned by the data element that uses it.

The meta-model elements of the Program Elements Layer uses the following naming conventions (whenever practical):

- suffix “Element” - usually designates a generic meta-model element.
- suffix “Type” - designates a meta-model element representing some datatype.
- suffic “Unit” - designates a concrete meta-model element.

12 Code Package

12.1 Overview

The Code package defines a set of meta-model elements whose purpose is to represent implementation level program elements and their associations. It is determined by one or more programming languages used in the design of the given existing software system. Code package includes meta-model elements, which represent common program elements supported by various programming languages, such as data types, data items, classes, procedures, macros, prototypes, and templates.

As a general rule, in a given KDM instance, each instance of the code meta-model element represents some programming language construct, determined by the programming language of the existing software system. Each instance of a code meta-model element corresponds to a certain region of the source code in one of the artifacts of the existing software system. Exceptions to this rule are:

- Instances of the CodeModel meta-model element that are parts of the KDM infrastructure. This meta-model element is a container for other code element instances.
- Instances of code element that explicitly represent certain abstractions provided by a programming language, such as primitive datatypes and predefined datatypes.

12.2 Organization of the Code Package

The Code package consists of the following 24 class diagrams.

1. CodeModel
2. CodeInheritances
3. Modules
4. ControlElements
5. DataElements
6. Values
7. PrimitiveTypes
8. EnumeratedTypes
9. CompositeTypes
10. DerivedTypes
11. Signature
12. DefinedTypes
13. ClassTypes
14. Templates
15. TemplateRelations
16. ClassRelations
17. TypeRelations
18. InterfaceRelations
19. PreprocessorDirectives
20. PreprocessorRelations
21. Comment
22. Visibility
23. VisibilityRelations
24. ExtendedCodeElements

The Code package depends on the following packages:

- Source
- Core
- kdm

12.3 CodeModel Class Diagram

The CodeModel class diagram follows the uniform pattern for KDM models and extends the KDM framework with specific meta-model elements related to implementation-level program elements and their associations.

The CodeModel diagram defines the following classes determined by the KDM model pattern:

- CodeModel – a class representing a model for CodeElement.
- AbstractCodeElement – a class representing an abstract parent class for all KDM entities that can be used to model code.
- AbstractCodeRelationship - a class representing an abstract parent of all KDM relationships that can be used to represent code.

The CodeModel diagram also defines several key abstract classes, which determine the KDM taxonomy for program elements:

- CodeItem
- ComputationalObject
- Datatype
- Module

The class diagram shown in Figure 12.1 captures these classes and their relations.

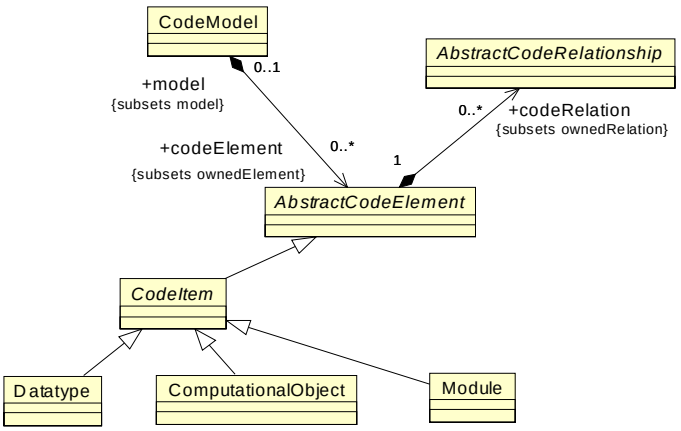

Figure 12.1 - CodeModel Class Diagram

12.3.1 CodeModel Class

The CodeModel is the specific KDM model that owns collections of facts about the existing software system such that these facts correspond to the Code domain. CodeModel is the only model of the Program Elements Layer of KDM. CodeModel follows the uniform pattern for KDM models.

Superclass

KDMModel

Associations

codeElement:AbstractCodeElement[0..*] {ordered}	The set of the top-level elements that are defined in this code model. The CodeModel element is the owner of such CodeElement. This property subsets the ownedElement property of KDMModel derived union.

Semantics

CodeModel is a container for code elements. The implementer shall arrange code elements into one or more code models. KDM import tools should not make any assumptions about the organization of code elements into code models.

12.3.2 AbstractCodeElement Class (abstract)

The AbstractCodeElement is an abstract class representing any generic determined by a programming language.

Superclass

KDMEntity

Associations

codeRelation:CodeRelation[0..*]	The set of code relations owned by this code model.
source: SourceRef[0..1]	Link to the physical artifact for the given code element.

Semantics

AbstractCodeElement is an abstract class that is used to constrain the owned elements of some KDM containers in the Code model.

12.3.3 AbstractCodeRelationship Class (abstract)

The AbstractCodeRelationship is an abstract class representing any relationship determined by a programming language.

Superclass

KDMRelationship

Semantics

AbstractCodeRelationship is an abstract class that is used to constrain the subclasses of KDMRelationship in the Code model.

12.3.4 CodeItem Class (abstract)

CodeItem class represents the named elements determined by the programming language (the so-called “symbols,” “definitions,” etc.). There are AbstractCodeElements that are not CodeItems, for example ActionElements that are defined in the Action package.

Superclass

AbstractCodeElement

Semantics

CodeItem is an abstract class that is used to constrain the owned elements of some KDM containers in the Code model.

12.3.5 ComputationalObject Class (generic)

ComputationalObject class represents the named elements determined by the programming language, which describe certain computational objects at the runtime, for example, procedures, and variables.

Superclass

CodeItem

Constraints

1. Instance of the ComputationalObject element should have at least one stereotype.

Semantics

ComputationalObject is a generic element with under specified semantics that can be used as an extension point to define new “virtual” meta-model elements that represent specific named control elements or data elements that do not fit into semantic categories of the concrete subclasses of ComputationalObject.

12.3.6 Datatype Class (generic)

Datatype class represents the named elements determined by the programming language that describes datatypes.

Superclass

CodeItem

Constraints

1. Instance of the Datatype element should have at least one stereotype.

Semantics

Datatype is a generic element with under specified semantics that can be used as an extension point to define new “virtual” meta-model elements that represent specific named datatypes that do not fit into semantic categories of the concrete subclasses of Datatype.

12.4 CodeInheritances Class Diagram

The CodeInheritances class diagram defines how classes of the Code package inherit from the Core package. The class diagram shown in Figure 12.2 captures these relations.

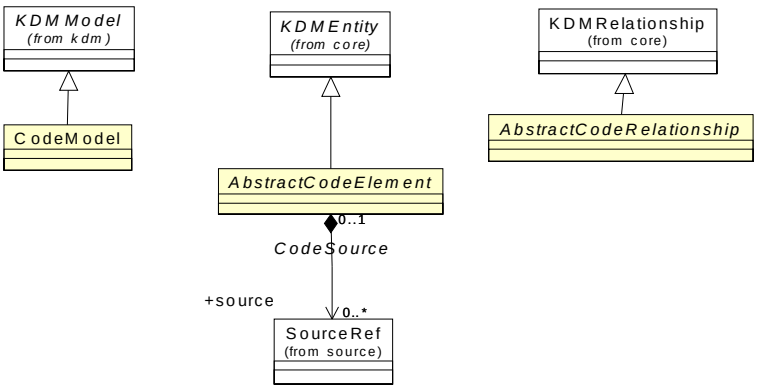

Figure 12.2 - CodeInheritances Class Diagram

Section I - Code Elements Representing Modules

12.5 Modules Class Diagram

The Modules class diagram defines meta-model elements that represent packaging aspects of programming languages, such as compilation units, shared files, and binary components. The class diagram shown in Figure 12.3 captures these classes and their associations.

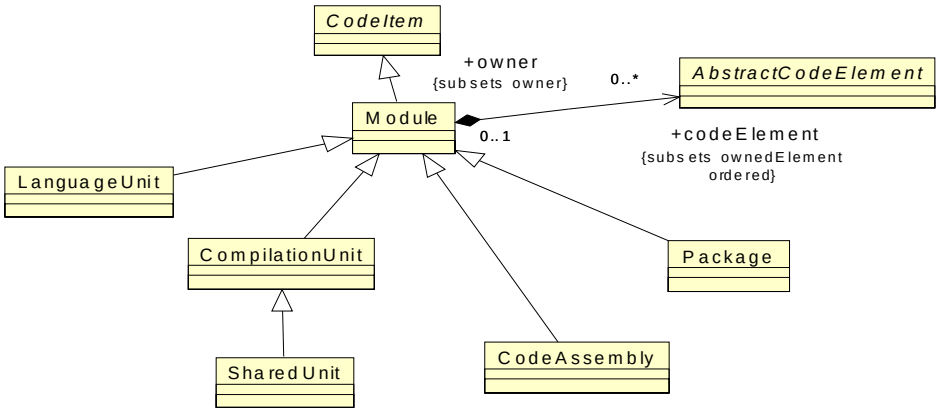

Figure 12.3 - Module Class Diagram

12.5.1 Module Class (generic)

The Module class is a generic KDM modeling element that represents an entire software module or a component, as determined by the programming language and the software development environment. A *module* is a discrete and identifiable program unit that contains other program elements and may be used as a logical component of the software system. Usually modules promote encapsulation (i.e., information hiding) through a separation between the interface and the implementation. In the context of representing existing software systems, modules provide the *context* for establishing the associations between the programming language elements that are owned by them, especially when the same logical

component of a software product line is compiled multiple times with different compilation options and linked into multiple executables. Instances of the Module class represent the logical containers for program elements determined by the programming language. Modules may be further related to other KDM items, for example to KDM Inventory items of the Inventory model; or to KDM deployment elements of the Platform model. In the situation of modeling a complex software product line, logical Modules may need to be duplicated, because the exact relationships determined by the particular software component may depend on the engineering context.

The Module is an abstract KDM container that provides a common parent for several concrete classes.

Superclass

CodeItem

Associations

codeElement:AbstractCodeElement[0..*] {ordered}	The list of owned CodeElement.

Constraints

1. Module class and its subclasses should not own SourceRef elements.
2. Code Model cannot directly own any code elements other than the subclasses of the Module class.
3. Every code element should be owned by some instance of the Module class or its subclasses.
4. Instance of the Module element should have at least one stereotype.
5. No other code element should own Module elements and its subclasses.

Semantics

Module is a logical container for program elements. Subclasses of Module element define semantically distinct flavors of Module, representing common categories of containers.

The implementer shall select an appropriate subclass of the Module element.

12.5.2 CompilationUnit Class

The CompilationUnit class is a meta-model element that represents a logical container that owns program elements. A *compilation unit* is a logical part of the existing software system that is sufficiently complete to be processed by the corresponding software development environment. Compilation unit is usually related to some artifact of the existing software system, for example, a physical source file. Compilation units are supported by the selected programming languages of the existing software system and as determined by the corresponding engineering process.

Superclass

Module

Semantics

CompilationUnit is a stand-alone named container for program elements. Usually a CompilationUnit corresponds to a SourceFile in the InventoryModel.

12.5.3 SharedUnit Class

The SharedUnit class is a meta-model element that represents a shared source file as supported by the selected programming languages of the existing software system and as determined by the engineering process.

Superclass

Module

Semantics

SharedUnit is a subclass of CompilationUnit, which emphasizes the ability of the program elements owned by the SharedUnit to be shared among stand-alone program elements through some form of inclusion mechanism.

12.5.4 LanguageUnit Class

The LanguageUnit class is a meta-model element, which represents predefined datatypes and other common elements determined by a particular programming language.

Superclass

Module

Constraints

1. PredefinedType class and its subclasses can only be contained in a LanguageUnit container.

Semantics

LanguageUnit is a logical container that owns definitions of primitive and predefined datatypes for a particular language, as well as other common elements for a particular programming language. LanguageUnit may or may not correspond to a SourceFile in the InventoryModel. Some of the predefined program elements are defined in standards system files. The implementer shall add such files to the InventoryModel. Primitive datatypes usually do not have any corresponding files, in this situation the LanguageUnit does not have a counterpart in the InventoryModel.

12.5.5 CodeAssembly Class

The CodeAssembly represents a logical container for the program elements that were built together (for example, compiled and linked into an executable, so that all variant selection during the compilation and static linking was resolved in a certain coordinated fashion). The same collection of logical entities has to be analyzed together. Same source files may produce a different logical model (for example, when compiled and linked for a different hardware platform, or for a different operating system). The CodeAssembly represents a collection of entities that have been analyzed together. A different variant of the conceptual family of software systems (even involving same compilation units) may need to be cloned into a separate CodeAssembly.

Superclass

Module

Semantics

CodeAssembly is a logical container that provides the context for entities and relationships for a collection of program elements. CodeAssembly may correspond to ExecutableFile elements of the InventoryModel. CodeAssembly may contain the so-called custom initialization block that refers to the initialization blocks of contained CompilationUnit elements.

12.5.6 Package Class

The Package class is a subtype for Module that logical collections of program elements, as directly supported by some programming languages, such as Java.

Superclass

Module

Semantics

A Package is a logical container for program elements as well as Modules. Packages can be nested.

Section II - Code Elements Representing Computational Objects

12.6 ControlElements Class Diagram

The ControlElements class diagram defines basic meta-model elements to represent callable computational objects, such as procedures, functions, methods, etc. The class diagram shown in Figure 12.4 shows these classes and their relations.

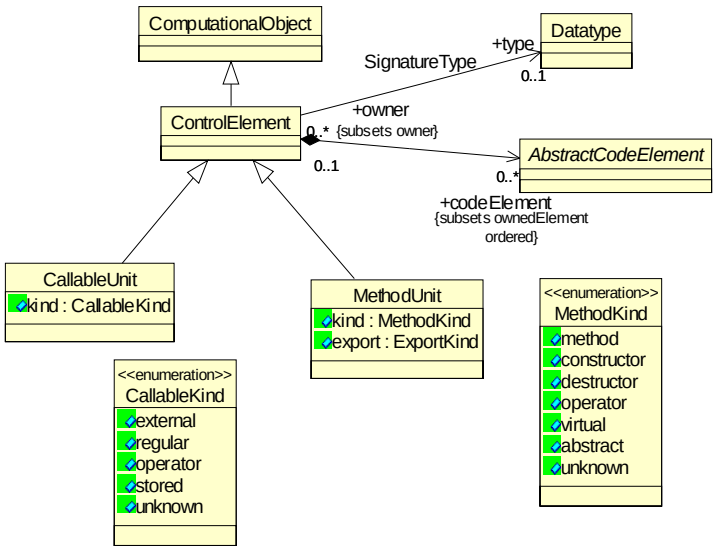

Figure 12.4 - ControlElements Class Diagram

12.6.1 ControlElement Class (generic)

The ControlElement class is a common superclass that defines attributes for callable code elements. In the meta-model it has the role of an endpoint for some KDM relations.

Superclass

ComputationalObject

Attributes and Associations

type:Datatype[0..1]	Optional association to the datatype of this control element.
codeElement:AbstractCodeElement[0..*] {ordered}	Represents owned code elements, such as local definitions and actions.

Constraints

1. ControlElement should have at least one stereotype.
2. ControlElement should own a Signature.

Semantics

ControlElement is a generic element with under specified semantics that can be used as an extension point to define new "virtual" meta-model elements that represent specific named control constructs that do not fit into semantic categories of the concrete subclasses of ControlElement.

ControlElement represents named items of the software system that describe certain behavior that can be performed by demand, through the invocation mechanism, such as a call-return mechanism, directly supported by many processor units and high-level programming languages.

ControlElement owns other program elements, which can include nested ControlElements.

12.6.2 CallableUnit Class

The CallableUnit represents a basic stand-alone element that can be called, such as a procedure or a function.

Superclass

ControlElement

Attributes

kind:CallableKind	indicator of the kind of the callable unit

Semantics

A CallableUnit represents a named unit of behavior that can be invoked through a call-return mechanism. This is a subclass of a ControlElement. From the runtime perspective, a CallableUnit element represents a single computational object, which is identified directly (using the name) or indirectly (using a reference). More precisely, the call-return mechanism implies an invocation stack, since a CallableUnit may call itself, and there may be multiple instances of the behavior represented by the CallableUnit, at various stages of completion, each corresponding to an entry in the invocation stack.

A CallableUnit represents global or local procedures and functions.

12.6.3 CallableKind Data Type (enumerated)

CallableKind enumerated data type specifies some common properties of the CallableUnit.

Literal values

regular	specifies a regular definition of a procedure or function
external	specifies an external procedure (a prototype, definition is elsewhere)
operator	specifies a definition of an operator
stored	specifies a stored procedure in DataModel
unknown	properties are unknown

12.6.4 MethodUnit Class

The MethodUnit represents member functions owned by a ClassUnit.

Superclass

CallableElement

Attributes

kind:MethodKind	Indicator of the kind of the method represented by this element.
export: ExportKind	Represents the visibility of the method (public, private, protected).

Semantics

The MethodUnit represents member functions owned by a ClassUnit, including user-defined operators, constructors, and destructors.

From the runtime perspective, each MethodUnit element represents a computational object that exists in the context of some class instance, therefore there exists multiple instances of such objects, each identified by the method name as well as the reference to the corresponding class instance. A class instance is identified either directly (by name) or indirectly (by reference).

12.6.5 MethodKind data type (enumeration)

MethodKind enumerated data type defines additional specification of the kind of method, defined by a MethodUnit model element.

Literal Values

method	The MethodUnit represents a regular member function.
constructor	The MethodUnit represents a constructor.
destructor	The MethodUnit represents a destructor.
operator	The MethodUnit represents an operator.
virtual	The MethodUnit represents a virtual method.

abstract	The MethodUnit represents an abstract method or member of an Interface.
unknown	The kind of the MethodUnit is none of the above.

Example (C language)

```
int main(int argc, char* argv[]) {
       printf("Hello, World\n");
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
       xmlns:source="http://schema.omg.org/spec/KDM/1.2/source"
name="HelloWorld Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel" name="HelloWorld">
    <codeElement xmi:id="id.1" xmi:type="code:CompilationUnit" name="hello.c">
      <codeElement xmi:id="id.2" xmi:type="code:CallableUnit"
                   name="main" type="id.5" kind="regular">
        <source xmi:id="id.3" language="C" snippet="int main(int argc, char* argv[]) {}"/>
        <entryFlow xmi:id="id.4" to="id.12" from="id.2"/>
        <codeElement xmi:id="id.5" xmi:type="code:Signature" name="main">
          <source xmi:id="id.6" snippet="int main(int argc, char * argv[]);"/>
          <parameterUnit xmi:id="id.7" name="argc" type="id.25" pos="1"/>
          <parameterUnit xmi:id="id.8" name="argv" type="id.9" pos="2">
            <codeElement xmi:id="id.9" xmi:type="code:ArrayType">
              <itemUnit xmi:id="id.10" type="id.19"/>
            </codeElement>
          </parameterUnit>
          <parameterUnit xmi:id="id.11" type="id.25" kind="return"/>
        </codeElement>
        <codeElement xmi:id="id.12" xmi:type="action:ActionElement" name="a1" kind="Call">
          <source xmi:id="id.13" language="C" snippet="printf(&amp;quot;Hello, World!\n&amp;quot;);"/>
          <codeElement xmi:id="id.14" xmi:type="code:Value"
             name="&amp;quot;Hello, World!\n&amp;quot;" type="id.19"/>
          <actionRelation xmi:id="id.15" xmi:type="action:Reads" to="id.14" from="id.12"/>
          <actionRelation xmi:id="id.16" xmi:type="action:Calls" to="id.20" from="id.12"/>
          <actionRelation xmi:id="id.17" xmi:type="action:CompliesTo"
                          to="id.20" from="id.12"/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.18" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.19" xmi:type="code:StringType" name="char *"/>
      <codeElement xmi:id="id.20" xmi:type="code:CallableUnit" name="printf" type="id.21">
        <codeElement xmi:id="id.21" xmi:type="code:Signature" name="printf">
          <parameterUnit xmi:id="id.22" name="" type="id.25" kind="return" pos="0"/>
          <parameterUnit xmi:id="id.23" name="format" type="id.19" pos="1"/>
          <parameterUnit xmi:id="id.24" name="arguments" kind="variadic" pos="2"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.25" xmi:type="code:IntegerType" name="int"/>
    </codeElement>
  </model>
  <model xmi:id="id.26" xmi:type="source:InventoryModel" name="HelloWorld">
    <inventoryElement xmi:id="id.27" xmi:type="source:SourceFile"
                      name="hello.c" language="C"/>
  </model>
</kdm:Segment>
```

12.7 DataElements Class Diagram

The DataElements class diagram defines meta-model constructs to represent the named data items of existing software systems (for example, global and local variables, record files, and formal parameters). The class diagram at Figure 12.5 shows these classes and their associations.

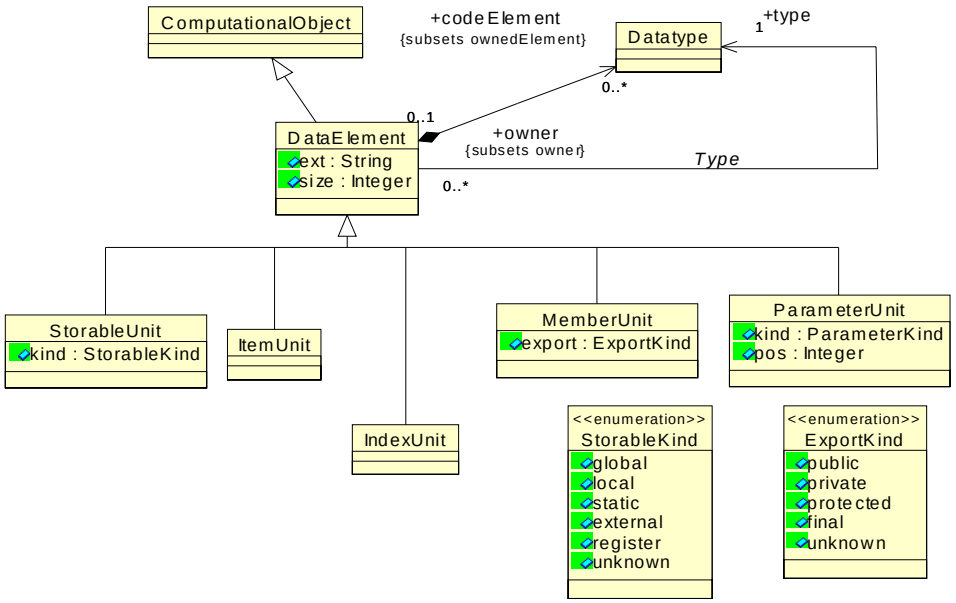

Figure 12.5 - DataElement Class Diagram

12.7.1 DataElement Class (generic)

The DataElement class is a generic modeling element that defines the common properties of several concrete classes that represent the named data items of existing software systems (for example, global and local variables, record files, and formal parameters). KDM models usually use specific concrete subclasses. The DataElement class itself is a concrete class that can be used as an extended code element, with a certain stereotype. As an extended element DataElement is more specific than CodeElement.

Superclass

ComputationalObject

Attributes

ext:String	Optional extension representing the original representation of the data element.
size: Integer	Specifies the optional constraint on the number of elements any value of the storable element may contain according to the semantics of the base datatype. Size attribute corresponds to the maximum-size bound in a size-subtype of the base datatype.

Associations

codeElement:Datatype[0..*]	Anonymous datatypes used in the definition of the datatype of the current DataElement.
type:Datatype[1]	The datatype of the DataElement that describes the values of the DataElement.

Semantics

DataElement represents computational objects of the existing software system that are associated with a value of a particular datatype. DataElement class is an extended meta-model element, that can be used to represent variables of an existing software system, that do not fit into more precise semantics of the subclasses of DataElement.

Constraints

1. DataElement class should have at least one Stereotype.

12.7.2 StorableUnit Class

StorableUnit class is a concrete subclass of the StorableElement class that represents variables of the existing software system.

Superclass

DataElement

Attribute

kind:StorableKind	Optional attribute that specifies the common details of a StorableUnit (see StorableKind enumeration datatype).

Semantics

StorableUnit represents a variable of existing software system - a computational object to which different values of the same datatype can be associated at different times. From the runtime perspective, a StorableUnit element represents a single computational object, which is identified either directly (by name) or indirectly (by reference).

StorableUnit represents both global and local variables.

12.7.3 StorableKind data type (enumeration)

StorableKind enumeration data type defines several common properties of a StorableUnit related to their life-cycle, visibility, and memory type.

Literal values

global	specifies a global variable
local	specifies a local variable
static	specifies a global variable with restricted scope

external	specifies an external variable (a prototype)
register	specifies a temporary variable
unknown	properties are unknown

12.7.4 ExportKind data type (enumeration)

ExportKind enumeration data type defines several common properties of a MemberUnit and MethodUnit related to their visibility and other properties.

Literal values

public	specifies a public member or method
private	specifies private member or method
protected	specifies a protected member or method
final	specifies final member or method
unknown	properties are unknown

12.7.5 ItemUnit Class

ItemUnit class is a concrete subclass of the DataElement class that represents anonymous data items that are parts of complex datatypes; for example, record fields, pointers, and arrays. Instances of ItemUnit class are endpoints of KDM data relations that describe access to complex datatypes.

Superclass

DataElement

Semantics

An ItemUnit represents a data element that exists in the context of another data element. From the runtime perspective, each ItemUnit represents a family of data elements, each of which is identified not only by the identity of the ItemUnit, but also by the identity of the owner element.

12.7.6 IndexUnit Class

IndexUnit class is a concrete subclass of the DataElement class that represents an index of an array datatype. Instances of IndexUnit class are endpoints of KDM data relations that describe access to arrays.

Superclass

DataElement

Semantics

IndexUnit represents an index of an ArrayType. IndexUnit is an optional element.

12.7.7 MemberUnit Class

MemberUnit class is a concrete subclass of the DataElement class that represents a member of a class type. Instances of MemberUnit class are endpoints of KDM data relations that describe access to classes. MemberUnit is similar to an ItemUnit. The difference between an ItemUnit and a MemberUnit is that an ItemUnit usually represents a part of a certain existing computational object, while the computational object corresponding to a MemberUnit is usually determined by the class instance. In case of accessing structures via pointers, this distinction becomes more subtle. MemberUnit defines some additional attributes.

Superclass

DataElement

Attributes

export:ExportKind	Represents the visibility of the member (public, private, protected).

Constraints

1. MemberUnit can be owned only by a ClassUnit.

Semantics

MemberUnit represents a member of a class. From the runtime perspective, each MemberUnit element represents a family of computational objects, each of which is a part of some class instance. Each MemberUnit is identified by the name of the MemberUnit, as well as by the direct or indirect identity of the corresponding class instance.

12.7.8 ParameterUnit Class

ParameterUnit class is a concrete subclass of the DataElement class that represents a formal parameter; for example, a formal parameter of a procedure. ParameterUnits are owned by the Signature element. Instances of ParameterUnit class are endpoints of KDM data relations that describe access to formal parameters.

Superclass

DataElement

Attributes

kind:ParameterKind	Optional attribute defining the parameter passing convention for the attribute.
pos:Integer	Position of the attribute in the signature.

Constraints

1. Return parameter of a signature does not have a pos attribute.
2. Return ParameterUnit is a signature should have a kind=”return.”
3. There can be at most one ParameterUnit within a certain Signature with a return kind.

Semantics

ParameterUnit is a data element that from the runtime perspective represents a computational object that exists in the context of an instance of some ControlElement “in the process of execution.”

Instances of ParameterUnit class are owned by instances of a Signature class. ParameterUnits within a Signature are ordered. The value of the pos attribute of a ParameterUnit should correspond to the position of the parameter in the Signature. The return ParameterUnit is distinguished by the value of the kind attribute. To represent signatures of programming languages that allow named parameters (binding of actual parameters by name rather than by a position), the producer of the KDM model is responsible for computing correct positions of the named parameters and determining appropriate ParameterUnits as targets for relations to named parameters. ParameterKind enumeration datatype is described in “Signature Class Diagram” on page 95.

12.8 ValueElements Class Diagram

ValueElements class diagram defines meta-model elements that represent data values, which are used in the artifacts of the existing software system.

The classes and associations of the ValueElements class diagram are shown at Figure 12.6.

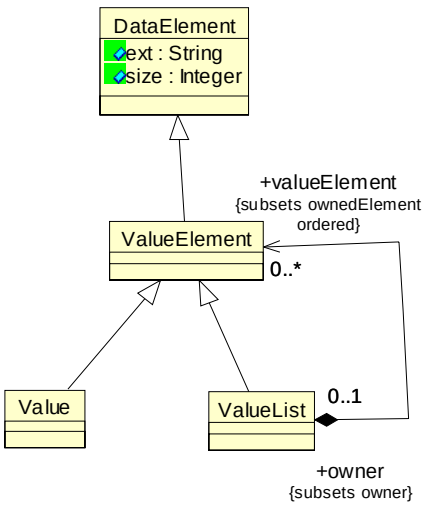

Figure 12.6 - ValueElements Class Diagram

12.8.1 ValueElement Class (generic)

ValueElement class is a generic meta-model element that represents values used in the artifacts of existing software systems. This class defines the common properties of the concrete subclasses, for which more precise semantics is provided. KDM model of an existing software system usually uses concrete subclasses of the ValueElement class.

Superclass

DataElement

Constraints

1. ValueElement and its subclasses should not have owned code elements.
2. ValueElement and its subclasses cannot be used as the target of relations Writes, and Addresses.
3. ValueElement class instance should have at least one Stereotype.

Semantics

A value element is a data element that represents a single value of the corresponding datatype.

ValueElement class and its subclasses correspond to ISO/IEC 11404 literals and values. The datatype of the value is represented by the type property (defined for its superclass DataElement class).

12.8.2 Value Class

Value class is a meta-model element that represents values used in the artifacts of existing software systems.

Superclass

ValueElement

Semantics

Value class corresponds to ISO/IEC 11404 literals of primitive types, such as boolean-literal, state-literal, enumerated-literal, character-literal, ordinal-literal, time-literal, integer-literal, rational-literal, scaled-literal, real-literal, void-literal, pointer-literal, bitstring-literal, string-literal.

The name attribute of the ValueClass represents the name or a string representation of the value.

12.8.3 ValueList Class

The ValueList class is a meta-model element that represents values of aggregated datatypes.

Superclass

ValueElement

Associations

valueElement:ValueElement[0..*]	component values

Semantics

A ValueList is a data element associated with a single value of some non-primitive datatype. The value of the complex datatype is represented as a tuple of values for each subcomponent of the complex datatype.

Value class corresponds to ISO/IEC 11404 values for aggregated datatypes such as choice-value, record-value, set-value, sequence-value, bag-value, array-value, table-value.

Section III - Code Elements Representing Datatypes

Data representation of KDM is aligned with ISO/IEC 11404 (General-Purpose Datatypes) standard. In particular, KDM provides distinct meta-model elements for “data elements” (for example, global and local variables, constants, record fields, parameters, class members, array items, and pointer base elements) and “datatypes.” Each data element has an association “type” to its datatype. KDM distinguishes:

- primitive datatypes (for example, Integer, Boolean),
- complex user-defined datatypes (for example, array, pointer, sequence), and
- named datatypes (for example, a class, a synonym type).

KDM meta-model elements corresponding to datatypes are subclasses of a generic class Datatype. KDM meta-model elements corresponding to data elements are subclasses of a generic class DataElement.

KDM model elements represent existing artifacts determined by a programming language. KDM meta-model elements provide sufficient coverage for most common datatypes and data elements, common to programming languages. KDM also provides several powerful generic extensible elements that can be further used with stereotypes to represent uncommon situations.

In addition to the type association, KDM relationship “HasType” is used to track named datatypes. Anonymous datatypes can be owned by the data element that uses it.

Concrete examples of datatypes, data items, and the use of the type association, and the HasType relationship are provided further in the text of the specification.

12.9 PrimitiveTypes Class Diagram

The PrimitiveTypes class diagram defines meta-model elements that represent predefined types common to various programming languages. The classes and association of the PrimitiveTypes diagram are shown in Figure 12.7.

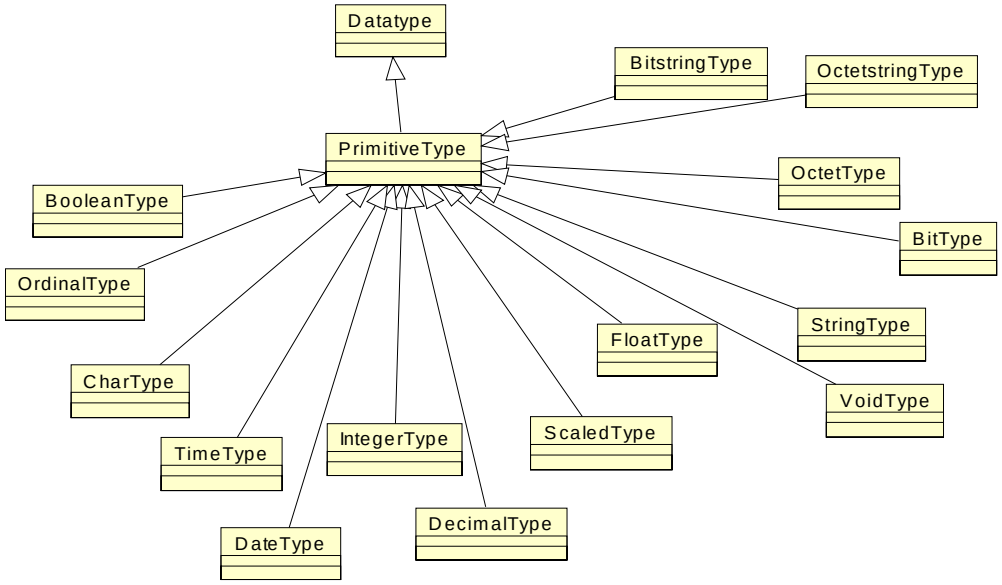

Figure 12.7 - PrimitiveTypes Class Diagram

12.9.1 PrimitiveType Class (generic)

The PrimitiveType is a generic meta-model element that represents primitive data types determined by various programming languages.

Superclass

Datatype

Constraints

1. PrimitiveType should have at least one stereotype.

Semantics

PrimitiveType element has under specified semantics. It can be used as an *extension point* to define new “virtual” meta-model elements to represent specific primitive types that do not fit into semantic categories defined by concrete subclasses of PrimitiveType class.

12.9.2 BooleanType Class

The BooleanType is a meta-model element that represents Boolean data types common to various programming languages. A Boolean is a mathematical datatype associated with two-valued logic.

Superclass

PrimitiveType

Semantics

The KDM BooleanType class corresponds to ISO/IEC 11404 Boolean datatype.

12.9.3 CharType Class

The CharType is a meta-model element that represents character data types common to various programming languages. Character is a family of datatypes whose value spaces are character-sets.

Superclass

PrimitiveType

Semantics

The KDM CharType class corresponds to ISO/IEC 11404 Character datatype.

12.9.4 OrdinalType Class

The OrdinalType class is a meta-model element that represents ordinal datatypes available in some programming languages. Ordinal is the datatype of the ordinal numbers, as distinct from the quantifying numbers (datatype Integer). Ordinal is the infinite enumerated type.

Superclass

PrimitiveType

Semantics

The KDM OrdinalType class corresponds to ISO/IEC 11404 Ordinal datatype.

12.9.5 DateType Class

The DateType is a meta-model element that represents built-in data types related to dates.

Superclass

PrimitiveType

Semantics

12.9.6 TimeType Class

The TimeType is a meta-model element that represents built-in data types related to time. Time is a family of datatypes whose values are points in time to various common resolutions: year, month, day, hour, minute, second, and fractions thereof.

Superclass

PrimitiveType

Semantics

The KDM TimeType class corresponds to ISO/IEC 11404 Time datatype. The interpretation of the details of the Time datatype (in particular the time unit) is outside of the scope of KDM. KDM analysis tools can be used to analyze the KDM representation of an existing system to systematically identify the detailed information regarding the details of the Time data items. The time-unit, and other attributes can be added to the data item of the TimeType.

12.9.7 IntegerType Class

The IntegerType is a meta-model element that represents integer data type common to various programming languages. Integer is the mathematical datatype comprising exact integer values.

Superclass

PrimitiveType

Semantics

The KDM IntegerType class corresponds to ISO/IEC 11404 Integer datatype.

12.9.8 DecimalType Class

The DecimalType is a meta-model element that represents decimal data types common to various programming languages.

Superclass

PrimitiveType

Semantics

The KDM DecimalType class corresponds to ISO/IEC 11404 Integer datatype.

12.9.9 ScaledType Class

The ScaledType is a meta-model element that represents fixed point data types common to various programming languages. Scaled is a family of datatypes whose value spaces are subsets of the rational value space, each individual datatype having a fixed denominator, but the scaled datatypes possess the concept of approximate value.

Superclass

PrimitiveType

Semantics

The KDM ScaledType class corresponds to ISO/IEC 11404 Scaled datatype.

12.9.10 FloatType Class

The FloatType is a meta-model element that represents float data types common to various programming languages. Float is a family of datatypes that are computational approximations to the mathematical datatype comprising the “real numbers.”

Superclass

PrimitiveType

Semantics

The KDM FloatType class corresponds to ISO/IEC 11404 Real datatype.

12.9.11 VoidType Class

The VoidType is a meta-model element that represents built-in “void” type defined in certain programming languages. Void is a datatype representing an object whose presence is syntactically or semantically required, but carries no information in a given instance.

Superclass

PrimitiveType

Semantics

The KDM VoidType class corresponds to ISO/IEC 11404 Void datatype.

12.9.12 StringType Class

The StringType is a meta-model element that represents string data type common to various programming languages. String is a datatype representing strings of characters from standard character-sets.

Superclass

PrimitiveType

Semantics

The KDM StringType class corresponds to ISO/IEC 11404 defined datatype Character string. The interpretation of the details of the character encoding of the StringType is outside of the scope of KDM. Multibyte character strings can be represented as StringType with a stereotype.

12.9.13 BitType Class

The BitType class is a meta-model element representing the bit datatype available in some programming languages. Bit is the datatype representing the binary digits “0” and “1.”

Superclass

PrimitiveType

Semantics

The KDM BitType class corresponds to ISO/IEC 11404 defined datatype Bit.

12.9.14 BitStringType Class

The BitStringType class is a meta-model element that represents bit string datatypes available in some programming languages. Bitstring is the datatype of variable-length strings of binary digits.

Superclass

PrimitiveType

Semantics

The KDM BitstringType class corresponds to ISO/IEC 11404 defined datatype Bit string.

12.9.15 OctetType Class

The OctetType class is a meta-model element that represents octet datatypes available in some programming languages. Octet is a datatype of 8-bit codes, as used for character-sets and private encodings.

Superclass

PrimitiveType

Semantics

The KDM OctetType class corresponds to ISO/IEC 11404 defined datatype Octet.

12.9.16 OctetStringType Class

The OctetStringType class is a meta-model element that represents octet string datatypes available in some programming languages. Octet string is a variable-length encoding using 8-bit codes.

Superclass

PrimitiveType

Semantics

The KDM OctetstringType class corresponds to ISO/IEC 11404 defined datatype Octet String.

12.10 EnumeratedTypes Class Diagram

The EnumeratedTypes class diagram defines meta-model elements that represent enumerated types, which are common to various programming languages. The classes and associations of the EnumeratedTypes diagram are shown in Figure 12.8.

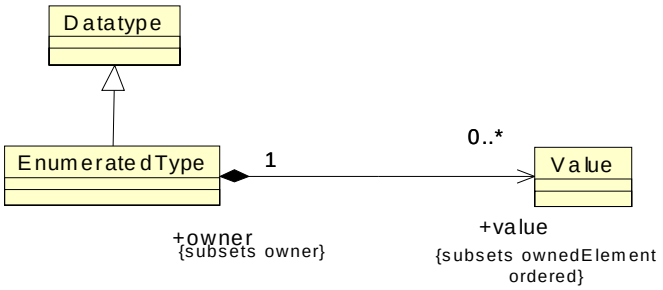

Figure 12.8 - EnumeratedTypes Class Diagram

12.10.1 EnumeratedType Class

The EnumeratedType is a meta-model element that represents user-defined enumerated data types. EnumeratedType datatype defines the set of enumerated literals. Enumerated datatype is a user-defined datatype, which has a finite number of distinguished values.

Superclass

Datatype

Associations

value:Value[0..*] {ordered}	The list of enumerated literals defined for the given EnumeratedType.

Semantics

EnumeratedType corresponds to ISO/IEC 11404 Enumerated and State families of datatypes. Enumerated datatype is a family of datatypes, each of which comprises a finite number of distinguished values having an intrinsic order. State is a family of datatypes, each of which comprises a finite number of distinguished but unordered values. KDM does not make distinction between these two families.

Values of the Enumerated and State datatypes are represented by a Value meta-model element that is owned by the EnumeratedType.

12.11 CompositeTypes Class Diagram

The CompositeTypes class diagram defines meta-model elements that.represent common composite datatypes provided by various programming languages; for example records, structures, and unions. Composite datatypes is a broad category of user-defined datatypes that includes situations in which the value of the datatype is made up of values of multiple component datatypes.

The classes and associations of the StructuredTypes diagram are shown in Figure 12.9.

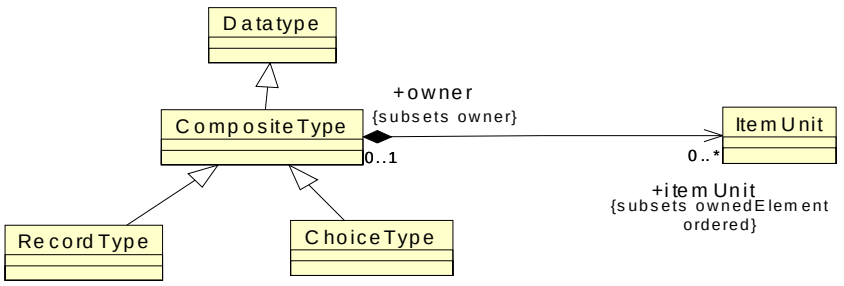

Figure 12.9 - CompositeTypes Class Diagram

12.11.1 CompositeType Class (generic)

The CompositeType is a meta-model element that represents user-defined composite datatypes, such as records, structures, and unions. This element is further subclassed by more specific KDM classes. CompositeType class defines common properties for its specific subclasses, each of which has distinct semantics. CompositeType class is a KDM

container. KDM models of existing software systems usually use the concrete subclasses of CompositeType class. CompositeType class itself is a concrete class and can be used as an extended meta-model element, with a stereotype. CompositeType class is a more specific meta-model element than CodeElement.

Superclass

Datatype

Associations

itemUnit:ItemUnit[0..*] {ordered}	The list of named items that represent components of the composite datatype; for example representing the individual fields of a record.

Constraints

1. CompositeType class should be used with a stereotype.

Semantics

CompositeType class corresponds to ISO/IEC 11404 generated datatypes each of whose values is made up of values of component datatypes. In particular, KDM CompositeType class corresponds to aggregate datatypes that involve a field list in their definition, and choice datatype. The Name attribute of each ItemUnit owned by the CompositeType represents the name of the field-type. The datatype of the field-type is represented by the type attribute of the ItemUnit.

CompositeType class is an extended meta-model element, that can be used to represent generated datatypes of an existing software system that do not fit into more precise semantics of the subclasses of CompositeType.

Any anonymous datatype used by an ItemUnit of the CompositeType should be owned by that ItemUnit.

12.11.2 ChoiceType Class

The ChoiceType class is a meta-model element that represents choice datatypes: user-defined datatypes in existing software systems, each of whose values is a single value from any of a set of alternative datatypes. An example of a choice datatype is a Pascal and Ada variant record, and a union in the C programming language. In the KDM representation, each alternative datatype is represented as an ItemUnit.

Superclass

CompositeType

Semantics

The ChoiceType corresponds to ISO/IEC 11404 choice generated datatype. KDM representation does not explicitly represent the field-identifier. Name attribute of each ItemUnit owned by the ChoiceType represents either the field-identifier of the alternative datatype or a single select item that identifies the variant. The datatype of the alternative is represented by the type attribute of the ItemUnit owned by the ChoiceType.

12.11.3 RecordType Class

The RecordType class is a meta-model element that represents record datatypes: user-defined datatypes in existing software systems, whose values are heterogeneous aggregations (tuples) of values of component datatypes, each aggregation having one value of each component datatype. Component datatypes are keyed by a fixed “field-identifier,” which is represented by the Name attribute of the ItemUnit owned by the RecordType. Examples of record datatypes include a structure in C, a record in Cobol.

Superclass

CompositeType

Semantics

The RecordType corresponds to ISO/IEC 11404 record aggregate datatype. The Name attribute of each ItemUnit owned by the RecordType represents the field-identifier. The datatype of the field is represented by the type attribute of the ItemUnit owned by the ChoiceType.

Example (Cobol)

```
01 StudentDetails.
   02 StudentId        PIC 9(7).
   02 StudentName.
      03 FirstName     PIC X(10).
      03 MiddleInitial PIC X.
      03 Surname       PIC X(15).
   02 DateOfBirth.
      03 DayOfBirth    PIC 99.
      03 MonthOfBirth  PIC 99.
      03 YearOfBirth   PIC 9(4).
   02 CourseCode       PIC X(4).

MOVE "Doyle" To Surname
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Record Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CompilationUnit">
      <codeElement xmi:id="id.2" xmi:type="code:StorableUnit"
                   name="StudentDetails" type="id.3">
        <codeElement xmi:id="id.3" xmi:type="code:RecordType" name="StudentDetails">
          <itemUnit xmi:id="id.4" name="StudentID" type="id.23" ext="PIC 9(7)"/>
          <itemUnit xmi:id="id.5" name="StudentName" type="id.6">
            <codeElement xmi:id="id.6" xmi:type="code:RecordType" name="StudentName">
             <itemUnit xmi:id="id.7" name="FirstName" type="id.24" ext="PIC X(10)" size="10"/>
              <itemUnit xmi:id="id.8" name="MiddleName" type="id.24" ext="PIC X" size="1"/>
              <itemUnit xmi:id="id.9" name="Surname" type="id.24" ext="PIC X(15)" size="15"/>
            </codeElement>
          </itemUnit>
          <itemUnit xmi:id="id.10" name="DateOfBirth">
            <codeElement xmi:id="id.11" xmi:type="code:RecordType" name="DateOfBirth">
              <itemUnit xmi:id="id.12" name="DayOfBirth" type="id.23" ext="PIC 99" size="2"/>
             <itemUnit xmi:id="id.13" name="MonthOfBirth" type="id.23" ext="PIC 99" size="2"/>
              <itemUnit xmi:id="id.14" name="YearOfBirth" type="id.23" ext="PIC 9(4)"
                                size="4"/>
            </codeElement>
          </itemUnit>
          <itemUnit xmi:id="id.15" name="CourseCode" type="id.24" ext="PIC X(4)" size="4"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.16" xmi:type="action:BlockUnit">
        <codeElement xmi:id="id.17" xmi:type="action:ActionElement">
          <codeElement xmi:id="id.18" xmi:type="code:Value"
                   name="&amp;quot;Doyle&amp;quot;" type="id.24"/>
```

```
          <actionRelation xmi:id="id.19" xmi:type="action:Addresses" to="id.2" from="id.17"/>
          <actionRelation xmi:id="id.20" xmi:type="action:Reads" to="id.18" from="id.17"/>
          <actionRelation xmi:id="id.21" xmi:type="action:Writes" to="id.9" from="id.17"/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.22" xmi:type="code:LanguageUnit" name="Cobol common definitions">
      <codeElement xmi:id="id.23" xmi:type="code:DecimalType"/>
      <codeElement xmi:id="id.24" xmi:type="code:StringType"/>
    </codeElement>
  </model>
</kdm:Segment>
```

12.12 DerivedTypes Class Diagram

The DerivedTypes class diagram defines meta-model elements that represent derived types, which are common to various programming languages. Examples of derived types include pointers, arrays, and sets. Derived datatypes is a broad category of user-defined datatypes that include situations, in which the value of the datatype is made up of values of a single component datatype, usually referred to as the element datatype. The classes and associations of the DerivedTypes diagram are shown in Figure 12.10.

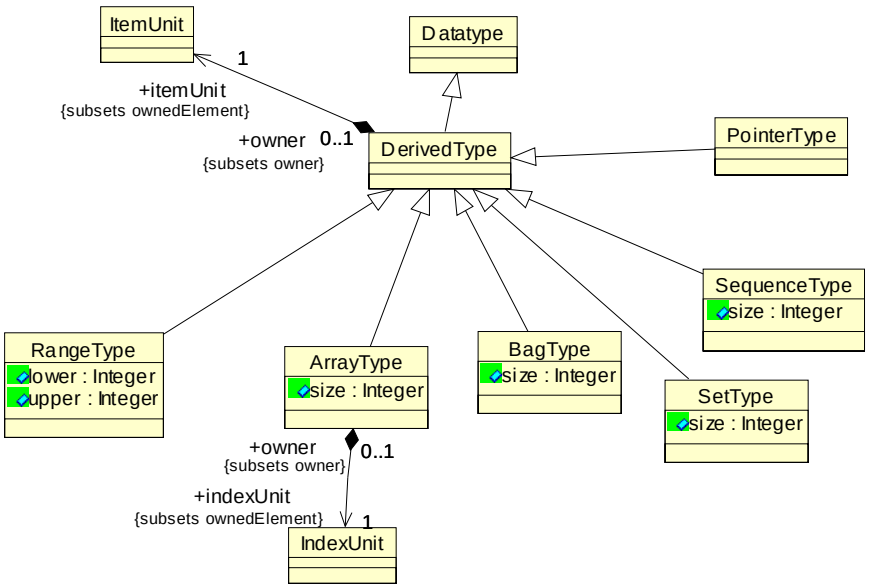

Figure 12.10 - DerivedTypes Class Diagram

12.12.1 DerivedType Class (generic)

DerivedType class defines common properties for its specific subclasses, each of which has distinct semantics. DerivedType class is a KDM container. KDM models of existing software systems usually use the concrete subclasses of DerivedType class. DerivedType class itself is a concrete class and can be used as an extended meta-model element, with a stereotype. DerivedType class is a more specific meta-model element than CodeElement.

Superclass

Datatype

Associations

itemUnit:ItemUnit[1]	The ItemUnit that represents the base class of the derived type.

Constraints

1. DerivedType class should be used with a stereotype.

Semantics

DerivedType class corresponds to several ISO/IEC 11404 aggregated datatypes, whose values are made up of values of a single component datatype. DerivedType class is an extended meta-model element, that can be used to represent aggregated datatypes with a single base datatype of an existing software system, that do not fit into more precise semantics of the subclasses of DerivedType. The name attribute of the ItemUnit can be omitted. The datatype of the element-type is represented by the type attribute of the ItemUnit owned by the DerivedType.

Any anonymous datatype used by ItemUnit of the DerivedType should be owned by that ItemUnit.

12.12.2 ArrayType Class

The ArrayType is a meta-model element that represents array datatypes.

Superclass

DerivedType

Attributes

size:Integer	The size of the array (the maximum number of elements).

Associations

indexUnit:IndexUnit[1]	The index of the array.

Semantics

ArrayType corresponds to ISO/IEC 11404 array datatype. The name attribute of the ItemUnit can be omitted if the element type is anonymous. The datatype of the element-type is represented by the type attribute of the ItemUnit owned by the ArrayType. The IndexItem represents the index of the array. The name attribute of the IndexUnit can be omitted.

KDM ArrayType supports a single index. Multidimensional arrays are represented by nested ArrayType elements, in which the top ArrayType represents the first (outer) dimension, and its ItemUnit has type ArrayType that represents the next (internal) dimension and so on.

Any anonymous datatype used by IndexUnit of the ArrayType should be owned by that IndexUnit.

12.12.3 PointerType Class

The PointerType is a meta-model element that represents pointer datatypes whose values constitute a means of reference to values of another datatype, designated the element datatype.

Superclass

DerivedType

Semantics

PointerType corresponds to ISO/IEC 11404 pointer generated datatype. From ISO perspective the pointer datatype is not an aggregated datatype, which leads to some mismatch with the semantics of the superclass. The Name attribute of the ItemUnit owned by the PointerType can be omitted. The datatype of the element-type is represented by the type attribute of the ItemUnit owned by the PointerType.

Example (C)

```
struct tlist {
    struct  tlist * next;
    int value;
} * phead, * pcurrent;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="LinkedList Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="action:BlockUnit">
      <codeElement xmi:id="id.2" xmi:type="code:StorableUnit"
                   name="phead" type="id.3" kind="unknown">
        <codeElement xmi:id="id.3" xmi:type="code:PointerType">
          <itemUnit xmi:id="id.4" type="id.5">
            <codeElement xmi:id="id.5" xmi:type="code:RecordType" name="tlist">
              <itemUnit xmi:id="id.6" name="next" type="id.3"/>
              <itemUnit xmi:id="id.7" name="value" type="id.8">
                <codeElement xmi:id="id.8" xmi:type="code:IntegerType" name="int"/>
              </itemUnit>
            </codeElement>
          </itemUnit>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.9" xmi:type="code:StorableUnit"
             name="pcurrent" type="id.3" kind="unknown"/>
    </codeElement>
  </model>
</kdm:Segment>
```

12.12.4 RangeType Class

RangeType is a meta-model element that represents user-defined subtypes of any ordered datatype by placing new upper and/or lower bounds on the value space.

Superclass

DerivedType

Attributes

lower: Integer	The optional lower boundary of the range.
upper: Integer	The optional upper boundary of the range.

Constraints

1. At least one boundary value attribute should be present.

Semantics

RangeType corresponds to ISO/IEC 11404 range subtype. From ISO perspective the range subtype is not an aggregated datatype, which leads to some mismatch with the semantics of the superclass. The Name attribute of the ItemUnit owned by the RangeType can be omitted. The datatype of the base type is represented by the type attribute of the ItemUnit owned by the RangeType.

When a boundary value attribute is omitted, this means that the corresponding value is unspecified.

12.12.5 BagType Class

BagType class is a meta-model element that represents bag types in existing software systems: the user-defined datatypes, whose values are collections of instances of values from the element datatype. Bag types allow multiple instances of the same value to occur in a given collection; the ordering of the value instances is not significant.

Superclass

DerivedType

Semantics

BagType corresponds to ISO/IEC 11404 bag aggregated datatype. The Name attribute of the ItemUnit owned by the BagType can be omitted. The datatype of the element type is represented by the type attribute of the ItemUnit owned by the BagType.

12.12.6 SetType Class

SetType is a meta-model element that represents set types in existing software systems: the user-defined datatypes, whose value space is the set of all subsets of the value space of the element datatype, with operations appropriate to the mathematical set.

Superclass

DerivedType

Semantics

SetType corresponds to ISO/IEC 11404 set aggregated datatype. The Name attribute of the ItemUnit owned by the SetType can be omitted. The datatype of the element type is represented by the type attribute of the ItemUnit owned by the SetType.

12.12.7 SequenceType Class

SequenceType class is a meta-model element that represents sequence types in existing software systems: the user-defined datatypes, whose values are ordered sequences of values from the element datatype. The ordering is imposed on the values and not intrinsic in the element datatype; the same value may occur more than once in a given sequence.

Superclass

DerivedType

Semantics

SequenceType corresponds to ISO/IEC 11404 sequence aggregated datatype. The Name attribute of the ItemUnit owned by the SequenceType can be omitted. The datatype of the element type is represented by the type attribute of the ItemUnit owned by the SequenceType.

12.13 Signature Class Diagram

The Signature class diagram defines meta-model elements, which represent the signature concept common to various programming languages. The classes and associations of the Signature diagram are shown in Figure 12.11.

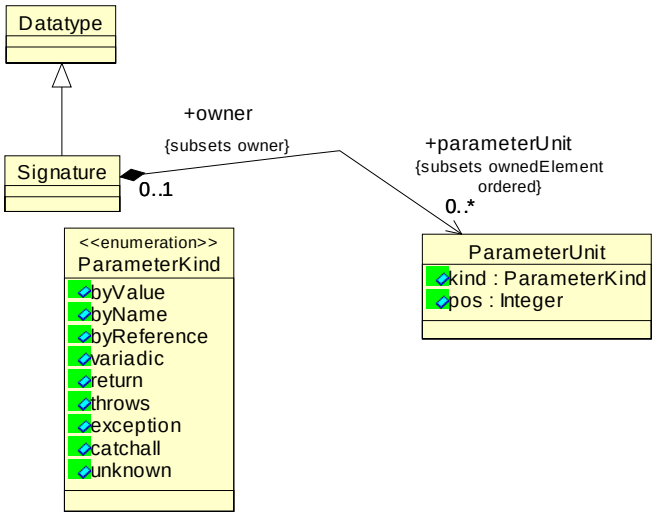

Figure 12.11 - Signature Class Diagram

12.13.1 Signature Class

The Signature is a meta-model element that represents the concept of a procedure signature, which is common to various programming languages.

Superclass

Datatype

Associations

parameterUnit:ParameterUnit[0..*]	The list of parameters of the current Signature.

Semantics

A Signature meta-model element has a dual role in KDM models. First, it corresponds to a procedure-type family of datatypes, corresponding to the procedure-type of ISO/IEC 11404 standard. Second, it corresponds to a specific data element as part of a computational object represented by a ControlElement. In this second sense, the Signature element corresponds to the mechanism of formal and actual parameters.

12.13.2 ParameterKind Enumeration Datatype

ParameterKind datatype defines the kind of parameter passing conventions.

Literals

byValue	parameter is passed by value
byName	parameter is passed by name
byReference	parameter is passed by reference
variadic	parameter is variadic
return	this is a return parameter
throws	parameter represents an exception thrown by the procedure
exception	parameter to a catch block
catchall	special parameter to a catch block
unknown	the parameter passing convention is unknown

Semantics

If the parameter kind is omitted, then the corresponding parameter is passed byValue. Return parameter is only distinguished by the parameter kind return value. If no parameters of a Signature have parameter kind value return, this means that the Signature does not define a return value.

12.14 DefinedTypes Class Diagram

DefinedTypes class diagram defines meta-model constructs to represent defined datatypes (datatypes defined by a type declaration). The capability of defining new type identifiers is supported in many programming languages.

The classes and associations involved in the definition of KDM DefinedTypes are shown in Figure 12.12.

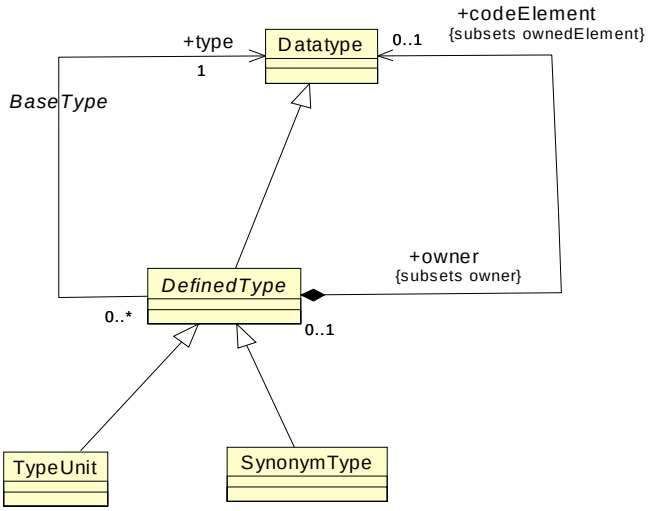

Figure 12.12 - DefinedTypes Class Diagram

12.14.1 DefinedType Class (abstract)

The DefinedType class is an abstract class that defines the common properties of several concrete classes that are used to represent type declarations in existing software systems.

Superclass

Datatype

Associations

codeElement:Datatype[0..*]	Anonymous datatypes used in the definition of the datatype.
type:Datatype[1]	The datatype of the DefinedType that describes the values of the corresponding datatype.

Semantics

DefinedType element represents a named element of existing software system, which corresponds to a user-defined datatype.

12.14.2 TypeUnit Class

The TypeUnit meta-model element represents the so-called new datatype declarations. New datatype declarations define the value-space of a new datatype, which is distinct from any other datatype.

Superclass

DefinedType

Semantics

TypeUnit corresponds to ISO/IEC 11404 New datatype declaration and New generator declarations.

12.14.3 SynonymUnit Class

The Synonym meta-model element represents the so-called renaming declarations. Renaming declarations declare the type name to be a synonym for another datatype.

Superclass

DefinedType

Semantics

SynonymUnit corresponds to ISO/IEC 11404 Renaming declarations.

12.15 ClassTypes Class Diagram

The ClassTypes class diagram defines meta-model elements that represent common composite datatypes provided by various programming languages. The classes and association of the ClassTypes diagram are shown in Figure 12.13.

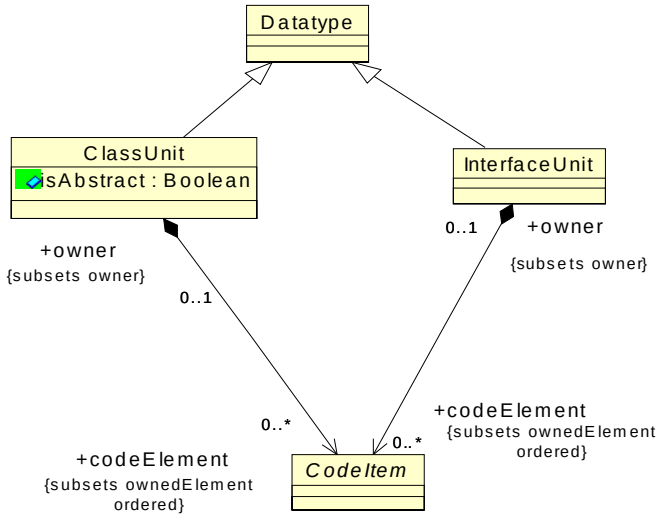

Figure 12.13 - ClassTypes Class Diagram

12.15.1 ClassUnit Class

The ClassUnit is a meta-model element that represents user-defined classes in object-oriented languages. A class datatype is a named datatype that represents a class: an ordered collection of named elements, each of which can be another CodeItem, such as a MemberUnit or a MethodUnit.

Superclass

Datatype

Attributes

isAbstract:Boolean	the indicator of an abstract class

Associations

codeElement:CodeItem[0..*]{ordered}	the list of class members

Semantics

ClassUnit is a named container for an ordered collection of named elements, each of which can be another CodeItem, such as a MemberUnit or a MethodUnit. Program elements owned by a ClassUnit may also include other (nested ClassUnits), internal datatype definitions, etc. From the runtime perspective, ClassUnit represents a family of computational objects, called class instances. MemberUnits and MethodUnits of a certain ClassUnit are identified both by the name of the member or method, as well as by a direct or indirect identification of the corresponding class instance.

12.15.2 InterfaceUnit Class

The InterfaceUnit is a meta-model element that represents the interface concept common to various programming languages.

In the meta-model InterfaceUnit is a subclass of Datatype. InterfaceUnit is a KDM container. InterfaceUnit owns a list of code items that represent data types as well as MethodUnits or CallableUnits. MethodUnit elements owned by an InterfaceUnit may be targets of Calls relations.

Superclass

Datatype

Associations

codeElement:CodeItem[0..*] {ordered}	The list of TypeElements that corresponds with the target Interface.

Semantics

InterfaceUnit is a logical container for code items. InterfaceUnit corresponds to a compile time description of the capabilities, that can be implemented by computational objects. InterfaceUnit owns datatype definitions as well as ControlElements, which in the representation of an existing software may be the targets of certain relationships, since the binding between the interface and the actual computational objects may occur at runtime.

12.16 Templates Class Diagram

The Templates class diagram provide basic meta-model constructs to define templates, parameters, instantiations of template and their relationships. Figure 12.14 shows these classes and their associations.

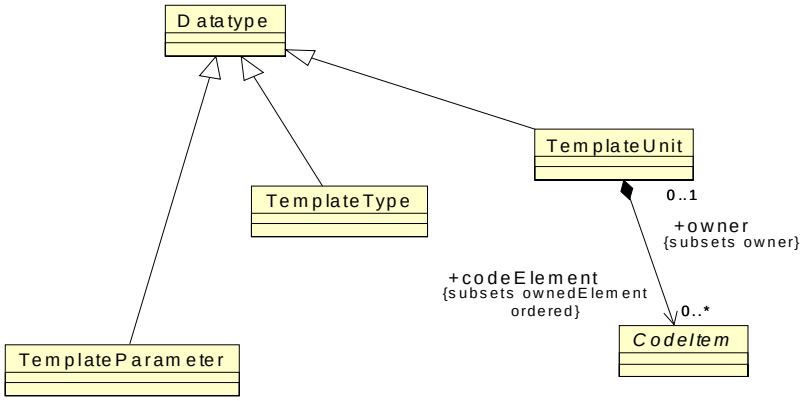

Figure 12.14 - Templates Class Diagram

12.16.1 TemplateUnit Class

The TemplateUnit is a meta-model element that represents parameterized datatypes, common to some programming languages; for example, Ada generics, Java generics, C++ templates.

Superclass

Datatype

Associations

codeElement:CodeItem[1]	template formal parameters and the base datatype or computational object

Constraints

1. TemplateParameter should be first in the list of code elements owned by the TemplateUnit.

Semantics

The TemplateUnit class corresponds to a type declaration with formal type parameters from the ISO/IEC 11404. TemplateUnit owns the corresponding datatype definition. Formal type parameters are represented by TemplateParameter elements owned by the TemplateUnit.

12.16.2 TemplateParameter Class

TemplateParameter is a meta-model element that represents parameters of a TemplateUnit. In the meta-model, TemplateParameter is a subclass of TypeElement.

Superclass

Datatype

Semantics

TemplateParameter represents a formal parameter of a type declaration with formal parameters (corresponding to ISO/IEC 11404). TemplateParameter is owned directly by a certain TemplateUnit. Correspondence between actual and formal parameters is positional.

12.16.3 TemplateType Class

TemplateType class is a meta-model element that represents references to parameterized datatypes. The TemplateType class owns the actual parameters to the datatype reference, represented by “ParameterTo” relationships. The TemplateType class also owns the “InstanceOf” relationship to the TemplateUnit that represents the referenced parameterized datatype. TemplateType has the role of a Datatype.

Superclass

Datatype

Constraints

1. TemplateType class should be the origin only to template relations “InstanceOf” and “ParameterTo.”

Semantics

The TemplateType class corresponds to a type-reference with actual type parameters to a type declaration with formal type parameters from the ISO/IEC 11404. The type declaration with formal parameters is represented by a TemplateUnit, which owns the corresponding datatype definition. Formal type parameters are represented by TemplateParameter elements owned by the TemplateUnit. The association between the type reference and the type declaration is represented by the “InstanceOf” relationship.

Relationship “ParameterTo” represents the actual type parameter. The association between the actual and formal type parameters is positional.

12.17 TemplateRelations Class Diagram

The TemplateRelations class diagram defines KDM relationships that are related to the concept of an template. Figure 12.15 shows these classes and their associations.

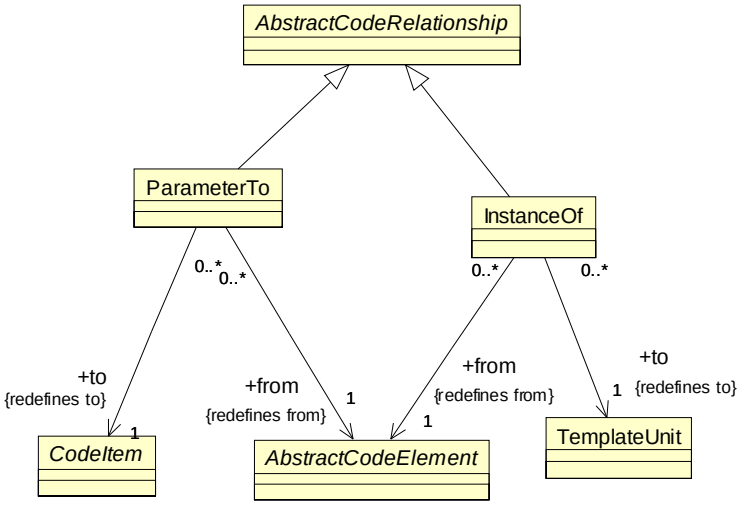

Figure 12.15 - TemplateRelations Class Diagram

12.17.1 InstanceOf Class

The InstanceOf is a meta-model element that represents "instantiation" relation between an AbstractCodeElement (for example, a ClassUnit) and a TemplateUnit. In the meta-model InstanceOf is a subclass of AbstractCodeRelationship.

Superclass

AbstractCodeRelationship

Associations

from:AbstractCodeElement[1]	The AbstractCodeElement that represents the instantiation of a template.
to:TemplateUnit[1]	The TemplateUnit that is being instantiated.

Constraints

1. The to- and from- endpoints of the relationship should be different.

Semantics

InstanceOf relationship represents an association between a reference to a parameterized datatype or a parameterized entity (for example, a generic method), to the corresponding declaration of the parameterized class.

12.17.2 ParameterTo Class

The ParameterTo is a meta-model element that represents an actual type parameter in the context of a reference to a parameterized entity. ParameterTo is "parametrization" relation between an AbstractCodeElement (for example, a TemplateType or an ActionElement) and a CodeItem.

Superclass

AbstractCodeRelationship

Associations

from:AbstractCodeElement[1]	the reference to the parameterized entity (the context of the actual type parameter)
to:CodeItem[1]	actual parameter to template instantiation

Constraints

1. ParameterTo relationship should be owned only by TemplateType or ActionElement.
2. The to- and from- endpoints of the relationship should be different.

Semantics

Reference to a parameterized datatype is represented by a TemplateType element. Another situation is an ActionElement that references a parameterized entity; for example, a call to a generic method. In this situation the ActionElement provides the context of the reference and owns the ParameterTo and InstanceOf relationships.

Example (Java)

```
class foo  {
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
   for (T o : a) {
          c.add(o);
          }
   }
void demo() {
       String[] sa = new String[100];
       Collection<String> cs = new ArrayList<String>();
       fromArrayToCollection(sa, cs);// T inferred to be String
       }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Template Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:ClassUnit" name="foo">
      <codeElement xmi:id="id.2" xmi:type="code:TemplateUnit"
                   name="fromArrayToCollection&lt;T>">
        <codeElement xmi:id="id.3" xmi:type="code:TemplateParameter" name="T"/>
        <codeElement xmi:id="id.4" xmi:type="code:MethodUnit"
                   name="fromArrayToCollection" type="id.6">
          <entryFlow xmi:id="id.5" to="id.14" from="id.4"/>
          <codeElement xmi:id="id.6" xmi:type="code:Signature">
            <parameterUnit xmi:id="id.7" name="a">
              <codeElement xmi:id="id.8" xmi:type="code:ArrayType">
                <itemUnit xmi:id="id.9" type="id.3"/>
              </codeElement>
            </parameterUnit>
            <parameterUnit xmi:id="id.10" name="c" type="id.11">
              <codeElement xmi:id="id.11" xmi:type="code:TemplateType"
                    name="Collection&lt;T1>">
                <codeRelation xmi:id="id.12" xmi:type="code:ParameterTo"
                           to="id.3" from="id.11"/>
               <codeRelation xmi:id="id.13" xmi:type="code:InstanceOf"
                           to="id.75" from="id.11"/>
              </codeElement>
            </parameterUnit>
```

```
    </codeElement>
    <codeElement xmi:id="id.14" xmi:type="action:ActionElement"
              name="a1" kind="Compound">
      <codeElement xmi:id="id.15" xmi:type="action:ActionElement"
              name="a1.1" kind="Call">
        <actionRelation xmi:id="id.16" xmi:type="action:Addresses"
                    to="id.7" from="id.15"/>
       <actionRelation xmi:id="id.17" xmi:type="action:Calls" to="id.81" from="id.15"/>
        <actionRelation xmi:id="id.18" xmi:type="action:Flow" to="id.19" from="id.15"/>
      </codeElement>
      <codeElement xmi:id="id.19" xmi:type="action:ActionElement"
                          name="a1.2" kind="Call">
        <codeElement xmi:id="id.20" xmi:type="code:StorableUnit"
                          name="t1" type="id.88" kind="register"/>
        <actionRelation xmi:id="id.21" xmi:type="action:Addresses"
                          to="id.40" from="id.19"/>
       <actionRelation xmi:id="id.22" xmi:type="action:Calls" to="id.83" from="id.19"/>
      <actionRelation xmi:id="id.23" xmi:type="action:Writes" to="id.20" from="id.29"/>
        <actionRelation xmi:id="id.24" xmi:type="action:Flow" to="id.25" from="id.19"/>
      </codeElement>
      <codeElement xmi:id="id.25" xmi:type="action:ActionElement"
                    name="1.3" kind="Condition">
       <actionRelation xmi:id="id.26" xmi:type="action:Reads" to="id.20" from="id.25"/>
        <actionRelation xmi:id="id.27" xmi:type="action:TrueFlow"
                    to="id.29" from="id.25"/>
        <actionRelation xmi:id="id.28" xmi:type="action:FalseFlow"
                    to="id.39" from="id.25"/>
      </codeElement>
      <codeElement xmi:id="id.29" xmi:type="action:ActionElement"
                    name="a1.4" kind="Call">
        <actionRelation xmi:id="id.30" xmi:type="action:Addresses"
                    to="id.40" from="id.29"/>
       <actionRelation xmi:id="id.31" xmi:type="action:Calls" to="id.82" from="id.29"/>
      <actionRelation xmi:id="id.32" xmi:type="action:Writes" to="id.44" from="id.29"/>
        <actionRelation xmi:id="id.33" xmi:type="action:Flow" to="id.34" from="id.29"/>
      </codeElement>
      <codeElement xmi:id="id.34" xmi:type="action:ActionElement"
                    name="a1.5" kind="Call">
        <actionRelation xmi:id="id.35" xmi:type="action:Addresses"
                    to="id.10" from="id.34"/>
       <actionRelation xmi:id="id.36" xmi:type="action:Reads" to="id.44" from="id.34"/>
       <actionRelation xmi:id="id.37" xmi:type="action:Calls" to="id.84" from="id.34"/>
        <actionRelation xmi:id="id.38" xmi:type="action:Flow" to="id.19" from="id.34"/>
      </codeElement>
     <codeElement xmi:id="id.39" xmi:type="action:ActionElement" name="1.6" kind="Nop"/>
      <codeElement xmi:id="id.40" xmi:type="code:StorableUnit"
                    name="iter" type="id.41" kind="register">
       <codeElement xmi:id="id.41" xmi:type="code:TemplateType" name="Iterator&lt;T1>">
          <codeRelation xmi:id="id.42" xmi:type="code:InstanceOf"
                          to="id.78" from="id.41"/>
          <codeRelation xmi:id="id.43" xmi:type="code:ParameterTo"
                          to="id.3" from="id.41"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.44" xmi:type="code:StorableUnit"
                    name="o" type="id.3" kind="local"/>
      <actionRelation xmi:id="id.45" xmi:type="action:Flow" to="id.15" from="id.14"/>
    </codeElement>
  </codeElement>
</codeElement>
<codeElement xmi:id="id.46" xmi:type="code:MethodUnit" name="demo" type="id.47">
  <codeElement xmi:id="id.47" xmi:type="code:Signature"/>
  <codeElement xmi:id="id.48" xmi:type="code:StorableUnit"
              name="sa" type="id.49" kind="local">
    <codeElement xmi:id="id.49" xmi:type="code:ArrayType" name="ar2">
      <itemUnit xmi:id="id.50" type="id.89"/>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.51" xmi:type="action:ActionElement" name="demo.1" kind="New">
    <codeElement xmi:id="id.52" xmi:type="code:Value" name="100" type="id.90"/>
    <actionRelation xmi:id="id.53" xmi:type="action:Reads" to="id.52" from="id.51"/>
    <actionRelation xmi:id="id.54" xmi:type="action:Creates" to="id.49" from="id.51"/>
    <actionRelation xmi:id="id.55" xmi:type="action:Writes" to="id.48" from="id.51"/>
    <actionRelation xmi:id="id.56" xmi:type="action:Flow"/>
  </codeElement>
  <codeElement xmi:id="id.57" xmi:type="code:StorableUnit"
```

```
                name="cs" type="id.58" kind="local">
        <codeElement xmi:id="id.58" xmi:type="code:TemplateType"
                name="Collection&lt;String>">
          <codeRelation xmi:id="id.59" xmi:type="code:ParameterTo" to="id.89" from="id.58"/>
           <codeRelation xmi:id="id.60" xmi:type="code:InstanceOf" to="id.75" from="id.58"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.61" xmi:type="action:ActionElement" name="demo.2" kind="New">
        <codeElement xmi:id="id.62" xmi:type="code:TemplateType"
                    name="ArrayList&lt;String>">
          <codeRelation xmi:id="id.63" xmi:type="code:ParameterTo" to="id.89" from="id.62"/>
           <codeRelation xmi:id="id.64" xmi:type="code:InstanceOf" to="id.85" from="id.62"/>
        </codeElement>
        <actionRelation xmi:id="id.65" xmi:type="action:Creates" to="id.62" from="id.51"/>
        <actionRelation xmi:id="id.66" xmi:type="action:Writes" to="id.57" from="id.61"/>
        <actionRelation xmi:id="id.67" xmi:type="action:Flow"/>
      </codeElement>
      <codeElement xmi:id="id.68" xmi:type="action:ActionElement" name="demo.3" kind="Call">
        <codeRelation xmi:id="id.69" xmi:type="code:InstanceOf" to="id.2" from="id.68"/>
        <codeRelation xmi:id="id.70" xmi:type="code:ParameterTo" to="id.89" from="id.68"/>
        <actionRelation xmi:id="id.71" xmi:type="action:Reads" to="id.48" from="id.68"/>
        <actionRelation xmi:id="id.72" xmi:type="action:Reads" to="id.57" from="id.68"/>
        <actionRelation xmi:id="id.73" xmi:type="action:Calls" to="id.4" from="id.68"/>
      </codeElement>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.74" xmi:type="code:LanguageUnit" name="Common Java datatypes">
    <codeElement xmi:id="id.75" xmi:type="code:TemplateUnit" name="Collection&lt;T>">
      <codeElement xmi:id="id.76" xmi:type="code:TemplateParameter" name="T"/>
      <codeElement xmi:id="id.77" xmi:type="code:ClassUnit" name="Collection"/>
    </codeElement>
    <codeElement xmi:id="id.78" xmi:type="code:TemplateUnit" name="Iterator&lt;T>">
      <codeElement xmi:id="id.79" xmi:type="code:TemplateParameter" name="T"/>
      <codeElement xmi:id="id.80" xmi:type="code:ClassUnit" name="Iterator">
        <codeElement xmi:id="id.81" xmi:type="code:MethodUnit"
                    name="iterator" kind="constructor"/>
        <codeElement xmi:id="id.82" xmi:type="code:MethodUnit" name="next"/>
        <codeElement xmi:id="id.83" xmi:type="code:MethodUnit" name="hasNext"/>
        <codeElement xmi:id="id.84" xmi:type="code:MethodUnit" name="add"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.85" xmi:type="code:TemplateUnit" name="ArrayList&lt;T>">
      <codeElement xmi:id="id.86" xmi:type="code:TemplateParameter" name="T"/>
      <codeElement xmi:id="id.87" xmi:type="code:ClassUnit" name="ArrayList"/>
    </codeElement>
    <codeElement xmi:id="id.88" xmi:type="code:BooleanType" name="Boolean"/>
    <codeElement xmi:id="id.89" xmi:type="code:StringType" name="String"/>
    <codeElement xmi:id="id.90" xmi:type="code:IntegerType" name="Integer"/>
  </codeElement>
 </model>
</kdm:Segment>
```

12.18 InterfaceRelations Class Diagram

The InterfaceRelations class diagram defines KDM relationships that represent the usages of a “declarations” by the corresponding “definition” code elements. The classes and associations of the InterfaceRelations diagram are shown in Figure 12.16.

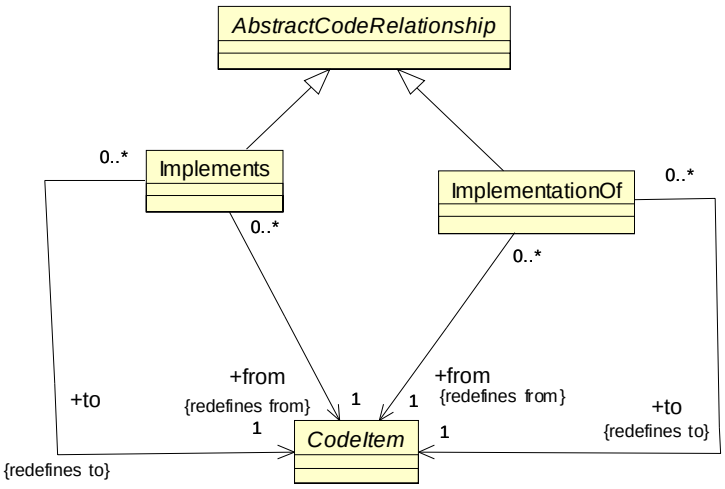

Figure 12.16 - InterfaceRelations Class Diagram

12.18.1 Implements Class

The Implements is a meta-model element that represents “implementation” association between a CodeItem (for example, a ClassUnit) and an InterfaceUnit. “Implements” relationship is similar to “Extends.” For example, Java “implements” construct can be represented by KDM “Implements” relationship.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	The CodeItem that implements a certain InterfaceUnit.
to:CodeItem[1]	The InterfaceUnit that is being implemented by CodeItem.

Constraints

1. The from- and to- endpoints should be different.

Semantics

See next section

12.18.2 ImplementationOf Class

The ImplementationOf is a meta-model element that represents “implementation” association between a CodeItem; for example, a MethodUnit and a particular “external” entity; for example, a MethodUnit owned by an InterfaceUnit. “ImplementationOf” relationship represents associations between a declaration and a definition of a computation object, common to various programming languages. While the “Implements” relationship is between entire containers (the target is an InterfaceUnit), the “ImplementationOf” relationship represents a broader range of situations:

- Particular MethodUnit of a ClassUnit that “Implements” an InterfaceUnit, is an “ImplementationOf” a particular MethodUnit, owned by that InterfaceUnit.
- A CallableUnit may be an “ImplementationOf” a CallableUnit with kind external, which represents the declaration (the prototype) of that CallableUnit.
- A StorableUnit may be an “ImplementationOf” a StorableUnit with kind external, which represents the external declaration of the StorableUnit, such as, for example, the “extern” construct in the C language.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	CodeItem that implements a certain “declaration.”
to:CodeItem[1]	“declaration” that is being implemented by the CodeItem.

Constraints

1. It is obligatory that either the origin of the ImplementationOf relationship is a ControlElement, and the target is a ControlElement or the origin is a DataElement and the target is a DataElement.
2. The kind attribute of the CodeItem at the origin of the ImplementationOf relationship should not be equal to “external.”
3. The kind attribute of the CodeItem at the target of the ImplementationOf relationship should be equal to “external” or “abstract.”
4. The from- and to- endpoints should be different.

Semantics

A “declaration” entity may be represented in KDM as a ComputationalObject (ControlElement or DataElement) with kind “external” or “abstract.” Kind “abstract” is used for the members of the InterfaceUnit. In case of a ControlElement, Signature represents the procedure type, but not the declaration entity itself.

If both the definition and the declaration of some computational object “foo” are available:

- The definition of “foo” may be the origin of the ImplementationOf relationship to the declaration of “foo.”
- For a certain action element that uses “foo,” the target of the KDM callable or data relations will be the definition of “foo.”
- The action element that uses “foo” may be the origin of a “CompliesTo” action relationship (defined at the InterfaceRelations class diagram of the Action package) to the declaration of “foo.”

If only the declaration of some computational object “bar” is available in the given context of capturing knowledge about the existing software system:

- For a certain action element that uses “bar,” the target of the KDM callable or data relations will be the declaration of “bar.”

- The action element that uses “bar” may be the origin of a “CompliesTo” action relationship (defined at the InterfaceRelations class diagram of the Action package) to the declaration of “bar.”

Declaration elements are usually owned by a certain SharedUnit.

In case of complex engineering process that involves multiple shared units, that are included into compilation units in complex ways, the existing software system may have multiple declarations for the same computational object, or even different computational objects with the same name but different properties that are used in different contexts. In this situation the above KDM mechanism that involves three relationships (the “real” usage, the “ImplementationOf,” and the “CompliesTo” relationships) can be used to detect subtle maintenance issues. For example, when for the same action element the target of the “ImplementationOf” relationship originating from the target of the “real” usage relationship, if different from the target of the “CompliesTo” relationship originating from the action element itself.

Example (Java):

```
package flip;
public interface iFlip {
       public int flip(int i);
}

package flip;
public class foo implements iFlip {
       public foo(){}
       public flip(int i) {
              return i * -1;
       }
}

package flip;
public class FlipClient {
       public static void main(String[] args) {
              foo f= new foo();
              iFlip g=(iFlip) f;
              f.flip(100);
       }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Interface Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:Package" name="flip">
      <codeElement xmi:id="id.2" xmi:type="code:ClassUnit" name="foo">
        <codeRelation xmi:id="id.3" xmi:type="code:Implements" to="id.21" from="id.2"/>
        <codeElement xmi:id="id.4" xmi:type="code:MethodUnit" name="flip" type="id.23">
          <codeRelation xmi:id="id.5" xmi:type="code:ImplementationOf"
                        to="id.22" from="id.4"/>
          <entryFlow xmi:id="id.6" to="id.10" from="id.4"/>
          <codeElement xmi:id="id.7" xmi:type="code:Signature" name="flip">
            <parameterUnit xmi:id="id.8" name="i" type="id.53"/>
            <parameterUnit xmi:id="id.9" type="id.53" kind="return"/>
          </codeElement>
          <codeElement xmi:id="id.10" xmi:type="action:ActionElement"
                        name="d1" kind="Multiply">
            <codeElement xmi:id="id.11" xmi:type="code:Value" name="-1" type="id.53"/>
            <codeElement xmi:id="id.12" xmi:type="code:StorableUnit"
                        name="t5" type="id.53" kind="register"/>
            <actionRelation xmi:id="id.13" xmi:type="action:Reads" to="id.8" from="id.10"/>
            <actionRelation xmi:id="id.14" xmi:type="action:Reads" to="id.11" from="id.10"/>
            <actionRelation xmi:id="id.15" xmi:type="action:Writes" to="id.12" from="id.10"/>
            <actionRelation xmi:id="id.16" xmi:type="action:Flow" to="id.17" from="id.10"/>
          </codeElement>
          <codeElement xmi:id="id.17" xmi:type="action:ActionElement" name="d2" kind="Return">
            <actionRelation xmi:id="id.18" xmi:type="action:Reads" to="id.12" from="id.17"/>
          </codeElement>
        </codeElement>
```

```
      <codeElement xmi:id="id.19" xmi:type="code:MethodUnit"
                   name="foo" type="id.20" kind="constructor">
        <codeElement xmi:id="id.20" xmi:type="code:Signature" name="foo"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.21" xmi:type="code:InterfaceUnit" name="IFlip">
      <codeElement xmi:id="id.22" xmi:type="code:MethodUnit"
                   name="flip" type="id.23" kind="abstract"/>
      <codeElement xmi:id="id.23" xmi:type="code:Signature" name="flip">
        <parameterUnit xmi:id="id.24" name="i" type="id.53" pos="1"/>
        <parameterUnit xmi:id="id.25" type="id.53" kind="return" pos="0"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.26" xmi:type="code:ClassUnit" name="Flipclient">
      <codeElement xmi:id="id.27" xmi:type="code:MethodUnit" name="main" type="id.29">
        <entryFlow xmi:id="id.28" to="id.35" from="id.27"/>
        <codeElement xmi:id="id.29" xmi:type="code:Signature" name="main">
          <parameterUnit xmi:id="id.30" name="args" type="id.31" pos="1">
            <codeElement xmi:id="id.31" xmi:type="code:ArrayType">
              <itemUnit xmi:id="id.32" name="args[]" type="id.54"/>
            </codeElement>
          </parameterUnit>
        </codeElement>
        <codeElement xmi:id="id.33" xmi:type="code:StorableUnit"
                   name="f" type="id.2" kind="local"/>
        <codeElement xmi:id="id.34" xmi:type="code:StorableUnit"
                   name="g" type="id.21" kind="local"/>
        <codeElement xmi:id="id.35" xmi:type="action:ActionElement" name="a1" kind="New">
          <actionRelation xmi:id="id.36" xmi:type="action:Creates" to="id.2" from="id.35"/>
          <actionRelation xmi:id="id.37" xmi:type="action:Writes" to="id.33" from="id.35"/>
          <actionRelation xmi:id="id.38" xmi:type="action:Flow" to="id.39" from="id.35"/>
        </codeElement>
        <codeElement xmi:id="id.39" xmi:type="action:ActionElement"
                   name="a2" kind="MethodCall">
          <actionRelation xmi:id="id.40" xmi:type="action:CompliesTo"
                   to="id.20" from="id.39"/>
          <actionRelation xmi:id="id.41" xmi:type="action:Addresses"
                   to="id.33" from="id.39"/>
          <actionRelation xmi:id="id.42" xmi:type="action:Calls" to="id.19" from="id.39"/>
          <actionRelation xmi:id="id.43" xmi:type="action:Flow" to="id.44" from="id.39"/>
        </codeElement>
        <codeElement xmi:id="id.44" xmi:type="action:ActionElement"
                   name="a3" kind="DynCast">
          <actionRelation xmi:id="id.45" xmi:type="action:Reads" to="id.33" from="id.44"/>
         <actionRelation xmi:id="id.46" xmi:type="action:UsesType" to="id.21" from="id.44"/>
          <actionRelation xmi:id="id.47" xmi:type="action:Writes" to="id.34" from="id.44"/>
          <actionRelation xmi:id="id.48" xmi:type="action:Flow" to="id.49" from="id.44"/>
        </codeElement>
        <codeElement xmi:id="id.49" xmi:type="action:ActionElement"
                   name="a4" kind="InterfaceCall">
          <actionRelation xmi:id="id.50" xmi:type="action:CompliesTo"
                   to="id.23" from="id.49"/>
          <actionRelation xmi:id="id.51" xmi:type="action:Addresses"
                   to="id.34" from="id.49"/>
          <actionRelation xmi:id="id.52" xmi:type="action:Calls" to="id.22" from="id.49"/>
        </codeElement>
      </codeElement>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.53" xmi:type="code:IntegerType" name="int"/>
  <codeElement xmi:id="id.54" xmi:type="code:StringType" name="String"/>
 </model>
</kdm:Segment>
```

12.19 TypeRelations Class Diagram

The TypeRelations class diagram defines meta-model elements that represent semantic associations between datatypes and data elements. The classes and associations of the TypeRelations diagram are shown in Figure 12.17.

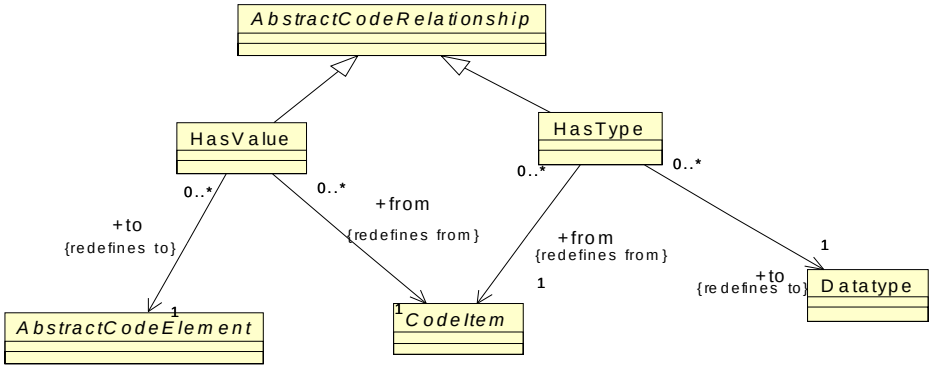

Figure 12.17 - TypeRelations Class Diagram

12.19.1 HasType Class

The HasType is a specific meta-model element that represents semantic relation between a data element and the corresponding type element.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	the source data element
to:Datatype[1]	the target datatype element

Constraints

1. The from- and to- endpoints should be different.

Semantics

HasType relationship represents an association between a code item that uses a certain user-defined type, and that datatype. Each data element has a direct association to its datatype. HasType relationship duplicates this information if the datatype is a named user-defined datatype (rather than an anonymous datatype or a primitive datatype) that allows a uniform representation of this information as a KDM relationship. In particular, this relationship can then be used in AggregatedRelationships.

12.19.2 HasValue Class

The HasValue is a specific meta-model element that represents semantic relation between a data element and its initialization element, which can be a data element or an action element for complex initializations that involve expressions. HasValue is an optional element that compliments the real initialization semantics by a sequence of action elements in the initialization code.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	the source data element
to:AbstractCodeElement[1]	the target AbstractCodeElement (datatype or action element)

Constraints

1. If the target of the HasValue is an ActionElement, then this ActionElement should have an outgoing Writes or Addresses relationship to the CodeItem that is the source of the HasValue relationship.

Semantics

HasValue relationship as an optional way to represent initialization. The target of the HasValue relationship can be a Value for simple initializations that involve constants, or Data Element for simple initializations that involve another data element, or an ActionElement that writes to the source element for complex initializations involving expressions.

In micro KDM initialization is represented by explicit initialization actions with appropriate control flow. Initialization actions should be owned by special initialization block units. Control flow semantics of initializations (especially for initializations of global and static data elements) is represented using the EntryFlow relationship. HasValue relationship does not represent control flow. It provides a convenient way to associate a data element with its value.

Example (C++)

```
/*----d.h---*/
class D {
private: int num;
public:
D(int x) { this->num=x; printf("Hello, this is %d\n", x); }
work() { printf("This is %d working\n", this->num);
};
/*---a.cpp---*/
#include "d.h"
int g1=0;
D d1(1);

/*---b.cpp--*/
#include "d.h"
extern D d1;
D d2(2);
main() {
   int l2=0;
   D * d3=new D(3);
   d1.work();
   d2.work();
   d3->work();
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="ClassD Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CodeAssembly">
      <entryFlow xmi:id="id.120" to="id.94" from="id.1"/>
      <codeElement xmi:id="id.2" xmi:type="code:CompilationUnit" name="a.cpp">
        <entryFlow xmi:id="id.121" to="id.10" from="id.2"/>
```

```
  <codeElement xmi:id="id.3" xmi:type="code:IncludeDirective" name="imp1">
    <codeRelation xmi:id="id.4" xmi:type="code:Includes" to="id.22" from="id.3"/>
  </codeElement>
  <codeElement xmi:id="id.5" xmi:type="code:StorableUnit" name="g1" type="id.105">
    <codeRelation xmi:id="id.6" xmi:type="code:HasValue" to="id.20" from="id.5"/>
  </codeElement>
  <codeElement xmi:id="id.7" xmi:type="code:StorableUnit" name="d1" type="id.23">
    <codeRelation xmi:id="id.8" xmi:type="code:HasType" to="id.23" from="id.7"/>
    <codeRelation xmi:id="id.9" xmi:type="code:ImplementationOf"
              to="id.47" from="id.7"/>
    <codeRelation xmi:id="id.124" xmi:type="code:HasValue" to="id.16" from="id.7"/>

  </codeElement>
  <codeElement xmi:id="id.10" xmi:type="action:BlockUnit" name="bi1" kind=”Init”>
    <entryFlow xmi:id="id.11" to="id.12" from="id.10"/>
    <codeElement xmi:id="id.12" xmi:type="action:ActionElement" name="i1" kind="Assign">
      <actionRelation xmi:id="id.13" xmi:type="action:Reads" to="id.20" from="id.12"/>
      <actionRelation xmi:id="id.14" xmi:type="action:Writes" to="id.5" from="id.12"/>
      <actionRelation xmi:id="id.15" xmi:type="action:Flow" to="id.16" from="id.12"/>
    </codeElement>
    <codeElement xmi:id="id.16" xmi:type="action:ActionElement" name="i2" kind="Calls">
      <actionRelation xmi:id="id.17" xmi:type="action:Reads" to="id.21" from="id.16"/>
      <actionRelation xmi:id="id.18" xmi:type="action:Calls" to="id.25" from="id.16"/>
      <actionRelation xmi:id="id.19" xmi:type="action:Writes" to="id.7" from="id.16"/>
    </codeElement>
    <codeElement xmi:id="id.20" xmi:type="code:Value" name="0"/>
    <codeElement xmi:id="id.21" xmi:type="code:Value" name="1"/>
  </codeElement>
</codeElement>
<codeElement xmi:id="id.22" xmi:type="code:SharedUnit" name="d.h">
  <codeElement xmi:id="id.23" xmi:type="code:ClassUnit" name="D">
    <codeElement xmi:id="id.24" xmi:type="code:MemberUnit"
              name="num" type="id.105" export="private"/>
    <codeElement xmi:id="id.25" xmi:type="code:MethodUnit" name="D">
      <entryFlow xmi:id="id.26" to="id.28" from="id.25"/>
      <codeElement xmi:id="id.27" xmi:type="code:Value"
              name="&amp;quot;Hello, this is %d\n&amp;quot;" type="id.111"/>
      <codeElement xmi:id="id.28" xmi:type="action:ActionElement"
              name="a4" kind="Assign">
        <actionRelation xmi:id="id.29" xmi:type="action:Reads" to="id.37" from="id.28"/>
       <actionRelation xmi:id="id.30" xmi:type="action:Writes" to="id.24" from="id.28"/>
        <actionRelation xmi:id="id.31" xmi:type="action:Flow" to="id.32" from="id.28"/>
      </codeElement>
      <codeElement xmi:id="id.32" xmi:type="action:ActionElement" name="a5" kind="Call">
        <actionRelation xmi:id="id.33" xmi:type="action:Reads" to="id.27" from="id.32"/>
        <actionRelation xmi:id="id.34" xmi:type="action:Reads" to="id.37" from="id.32"/>
       <actionRelation xmi:id="id.35" xmi:type="action:Calls" to="id.106" from="id.32"/>
      </codeElement>
      <codeElement xmi:id="id.36" xmi:type="code:Signature" name="D">
        <parameterUnit xmi:id="id.37" name="x" pos="1"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.38" xmi:type="code:MethodUnit" name="work">
      <codeElement xmi:id="id.39" xmi:type="code:Value"
              name="&amp;quot;This is %d working\n&amp;quot;"/>
      <codeElement xmi:id="id.40" xmi:type="action:ActionElement" name="a6" kind="Call">
        <actionRelation xmi:id="id.41" xmi:type="action:Reads" to="id.39" from="id.40"/>
        <actionRelation xmi:id="id.42" xmi:type="action:Reads" to="id.24" from="id.40"/>
       <actionRelation xmi:id="id.43" xmi:type="action:Calls" to="id.106" from="id.40"/>
      </codeElement>
    </codeElement>
  </codeElement>
</codeElement>
<codeElement xmi:id="id.44" xmi:type="code:CompilationUnit" name="b.cpp">
  <entryFlow xmi:id="id.122" to="id.87" from="id.44"/>
  <codeElement xmi:id="id.45" xmi:type="code:IncludeDirective" name="imp2">
    <codeRelation xmi:id="id.46" xmi:type="code:Includes" to="id.22" from="id.45"/>
  </codeElement>
  <codeElement xmi:id="id.47" xmi:type="code:StorableUnit"
              name="extern d1" kind="external"/>
  <codeElement xmi:id="id.48" xmi:type="code:CallableUnit" name="main">
    <entryFlow xmi:id="id.49" to="id.70" from="id.48"/>
    <codeElement xmi:id="id.50" xmi:type="code:StorableUnit" name="l2" type="id.105">
      <codeRelation xmi:id="id.51" xmi:type="code:HasValue" to="id.20" from="id.50"/>
    </codeElement>
    <codeElement xmi:id="id.52" xmi:type="code:StorableUnit" name="d2">
```

```
      <codeRelation xmi:id="id.53" xmi:type="code:HasType" to="id.23" from="id.52"/>
      <codeRelation xmi:id="id.125" xmi:type="code:HasValue" to="id.89" from="id.52"/>
    </codeElement>
    <codeElement xmi:id="id.54" xmi:type="code:StorableUnit" name="d3" type="id.55">
      <codeRelation xmi:id="id.126" xmi:type="code:HasValue" to="id.79" from="id.54"/>
      <codeElement xmi:id="id.55" xmi:type="code:PointerType">
        <itemUnit xmi:id="id.56" type="id.23">
          <codeRelation xmi:id="id.57" xmi:type="code:HasType" to="id.23" from="id.56"/>
        </itemUnit>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.58" xmi:type="action:ActionElement" name="a1" kind="Call">
      <actionRelation xmi:id="id.59" xmi:type="action:Calls" to="id.38" from="id.58"/>
     <actionRelation xmi:id="id.60" xmi:type="action:Addresses" to="id.7" from="id.58"/>
      <actionRelation xmi:id="id.61" xmi:type="action:CompliesTo"
                      to="id.47" from="id.58"/>
      <actionRelation xmi:id="id.62" xmi:type="action:Flow" to="id.63" from="id.58"/>
    </codeElement>
    <codeElement xmi:id="id.63" xmi:type="action:ActionElement" name="a2" kind="Call">
      <actionRelation xmi:id="id.64" xmi:type="action:Calls" to="id.38" from="id.63"/>
      <actionRelation xmi:id="id.65" xmi:type="action:Addresses"
                      to="id.52" from="id.63"/>
      <actionRelation xmi:id="id.66" xmi:type="action:Flow" to="id.67" from="id.63"/>
    </codeElement>
    <codeElement xmi:id="id.67" xmi:type="action:ActionElement" name="a3" kind="Call">
      <actionRelation xmi:id="id.68" xmi:type="action:Calls" to="id.38" from="id.67"/>
      <actionRelation xmi:id="id.69" xmi:type="action:Addresses"
                      to="id.56" from="id.67"/>
    </codeElement>
    <codeElement xmi:id="id.70" xmi:type="action:BlockUnit" name="bi2" kind=”Init”>
      <codeElement xmi:id="id.71" xmi:type="action:ActionElement"
                      name="i3" kind="Assign">
        <actionRelation xmi:id="id.72" xmi:type="action:Reads" to="id.20" from="id.71"/>
       <actionRelation xmi:id="id.73" xmi:type="action:Writes" to="id.50" from="id.71"/>
        <actionRelation xmi:id="id.74" xmi:type="action:Flow" to="id.79" from="id.71"/>
      </codeElement>
      <codeElement xmi:id="id.75" xmi:type="action:ActionElement" name="i4" kind="New">
        <actionRelation xmi:id="id.76" xmi:type="action:Creates"
                      to="id.23" from="id.75"/>
       <actionRelation xmi:id="id.77" xmi:type="action:Writes" to="id.54" from="id.75"/>
        <actionRelation xmi:id="id.78" xmi:type="action:Flow" to="id.79" from="id.75"/>
      </codeElement>
      <codeElement xmi:id="id.79" xmi:type="action:ActionElement"
                      name="i5" kind="MethodCall">
        <actionRelation xmi:id="id.80" xmi:type="action:Reads" to="id.85" from="id.79"/>
        <actionRelation xmi:id="id.81" xmi:type="action:Addresses"
                      to="id.54" from="id.79"/>
        <actionRelation xmi:id="id.82" xmi:type="action:Calls" to="id.25" from="id.79"/>
       <actionRelation xmi:id="id.83" xmi:type="action:Writes" to="id.56" from="id.79"/>
        <actionRelation xmi:id="id.84" xmi:type="action:Flow" to="id.58" from="id.79"/>
      </codeElement>
      <codeElement xmi:id="id.85" xmi:type="code:Value" name="3"/>
       <entryFlow xmi:id="id.86" to="id.71" from="id.70"/>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.87" xmi:type="action:BlockUnit" name="bi3" kind=”Init”>
    <entryFlow xmi:id="id.88" to="id.89" from="id.87"/>
    <codeElement xmi:id="id.89" xmi:type="action:ActionElement" name="i6" kind="Call">
      <actionRelation xmi:id="id.90" xmi:type="action:Reads" to="id.93" from="id.89"/>
      <actionRelation xmi:id="id.91" xmi:type="action:Calls" to="id.25" from="id.89"/>
      <actionRelation xmi:id="id.92" xmi:type="action:Writes" to="id.52" from="id.89"/>
    </codeElement>
    <codeElement xmi:id="id.93" xmi:type="code:Value" name="2" type="id.105"/>
  </codeElement>
</codeElement>
<codeElement xmi:id="id.94" xmi:type="action:BlockUnit" name="bi4" kind="Init">
 <entryFlow xmi:id="id.95" to="id.96" from="id.94"/>
 <codeElement xmi:id="id.96" xmi:type="action:ActionElement" name="i7" kind="Init">
    <entryFlow xmi:id="id.97" to="id.10" from="id.96"/>
    <actionRelation xmi:id="id.98" xmi:type="action:Flow" to="id.99" from="id.96"/>
 </codeElement>
 <codeElement xmi:id="id.99" xmi:type="action:ActionElement" name="i8" kind="Init">
    <entryFlow xmi:id="id.100" to="id.87" from="id.99"/>
    <actionRelation xmi:id="id.101" xmi:type="action:Flow" to="id.102" from="id.99"/>
 </codeElement>
 <codeElement xmi:id="id.102" xmi:type="action:ActionElement" name="i9" kind="Call">
```

```
          <actionRelation xmi:id="id.103" xmi:type="action:Calls" to="id.48" from="id.102"/>
       </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.104" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.105" xmi:type="code:IntegerType" name="int"/>
      <codeElement xmi:id="id.106" xmi:type="code:CallableUnit" name="printf" type="id.107">
        <codeElement xmi:id="id.107" xmi:type="code:Signature" name="printf">
          <parameterUnit xmi:id="id.108" type="id.105" kind="return" pos="0"/>
          <parameterUnit xmi:id="id.109" name="format" type="id.111" pos="1"/>
          <parameterUnit xmi:id="id.110" name="arguments" type="id.112"
                  kind="variadic" pos="2"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.111" xmi:type="code:StringType" name="char *"/>
      <codeElement xmi:id="id.112" xmi:type="code:VoidType"/>
    </codeElement>
  </model>
</kdm:Segment>
```

12.20 ClassRelations Class Diagram

The ClassRelations class diagram defines meta-model elements that represent semantic associations between datatypes. The classes and associations of the ClassRelations diagram are shown in Figure 12.18.

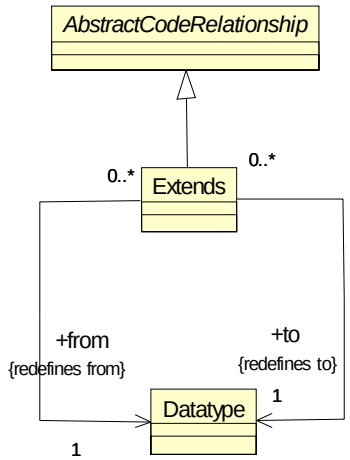

Figure 12.18 - ClassRelations Class Diagram

12.20.1 Extends Class

The Extends is a specific meta-model element that represents semantic relation between two classes, where one class (called a “child” class) extends another class (called its “parent” class) through inheritance, common to object-oriented languages.

Superclass

AbstractCodeRelationship

Associations

from:Datatype[1]	the child Class
to:Datatype[1]	the parent Class

Constraints

1. The from- and to- endpoints should be different.

Semantics

Extends relationship represents the associations between two datatypes in which the first datatype (called the “child” class) “subclasses” the second datatype (called the “parent” class) by inheriting the semantics and owned elements of the parent class.

Section IV - Code Elements representing Preprocessor Directives

A typical preprocessor allows the use of an embedded language in the source code written in some primary language. It is different than let’s say an HTML/Javascript program or programs with embedded assembler routines where there is more than one language but those don’t end up forming a modified version in a single language that is what gets compiled and executed (and that could be traced after, etc.). Example of the use of the preprocessor include Cobol copy replacing, Exec CICS, Exec SQL, 4GL user-defined language elements (such as MENTER macro/command in Dataflex).

The Exec SQL in Cobol example might be one of the best and easiest to look at. From a modeling/representation perspective the SQL statement itself is very much what we want to capture along with the various data elements and variables involved and their flow. However, an embedded SQL statement is expanded by the SQL pre-processor that generates some low-level code that will translate into a call to some library. By capturing this as generated code and correctly associating the elements, we can understand the program both from its SQL semantic, as well as from its execution realm. And our relationship between the “SQL language” elements to the “native” code elements that are generated is the glue tying things together. Trying to do the same analysis and operations by simply working with a couple of SourceRef would be very limiting indeed.

Preprocessor directive is a language/dialect on its own and it should be handled and treated as a first class citizen in KDM. However, the preprocessor support is an optional part of the code model, so that and the non-preprocessor-enabled L0 KDM tool can seamlessly ignore the embedded language and work only with the primary language. The implementer shall either:

1. Represent the embedded language in KDM and ignore the primary code that results from expansion of some preprocessor directives (provided this can lead to a meaningful model).
2. Ignore the embedded language and represent only the primary language in KDM.
3. Represent both the embedded and the generated primary code and relations between the two.

In order to achieve this goal, KDM modeling framework provides a strong separation between embedded language representation and primary language representation.

Preprocessor support in KDM allows conveying information that a certain code element appeared as the result of being:

- originally coded in the primary language
- included from another file by a preprocessor

- generated by a preprocessor as an expansion of an embedded language directive
- selected by satisfying appropriate conditions by the preprocessor

KDM provides the following modeling elements for representing preprocessor directives:

- PreprocessorDirective - representation of a generic preprocessor directive and a superclass for several other concrete preprocessor directives.
- MacroUnit -representation of macro definitions.
- MacroDirective - representation of an embedded language construct as distinguishable from the primary language construct. This is also known as a Macro Call.
- IncludeDirective - representation of an include directive of a preprocessor.
- ConditionalDirective - representation of a pre-processor conditional branch.

12.21 Preprocessor Class Diagram

The Preprocessor class diagram defines the meta-model elements to represent embedded language constructs and to support common modeling situations resulting from the use of a language preprocessor (for example, preprocessor defined as part of the C language, or the preprocessing capabilities of Cobol).

The class diagram in Figure 12.19 shows these classes and their associations.

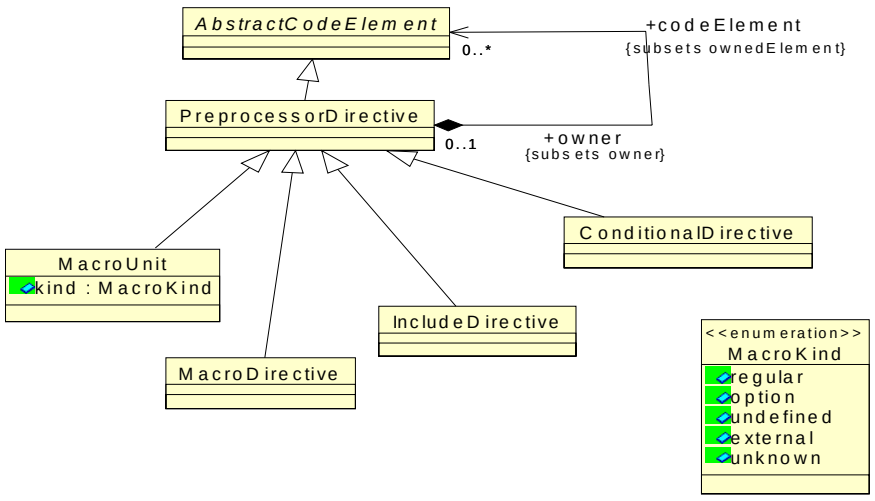

Figure 12.19 - Preprocessor Class Diagram

12.21.1 PreprocessorDirective Class (generic)

PreprocessorDirective is a generic meta-model element that represents preprocessor directives common to some programming languages (for example, the C language preprocessor capabilities). This class is extended by several concrete meta-model elements that represent several key directive types common to language preprocessors. KDM representations of existing systems are expected to use concrete subclasses of PreprocessorDirective, however this class

itself is a concrete meta-model element and can be used as an extended element with an appropriate stereotype to represent other types of preprocessing directives not covered by the standard subclasses. Semantics of preprocessor directives in KDM is described later in this section.

Superclass

AbstractCodeElement

Associations

codeElement:AbstractCodeElement[0..*]	This optional code element represents the content of the preprocessor directive.

Constraints

1. PreprocessorDirective should have a stereotype.

Semantics

From the KDM perspective, each preprocessor directive (an embedded language statement) is a container for code elements (possibly empty). KDM preprocessor support does not define any further special elements for semantic-rich representation of the embedded language directives. The implementer may provide additional information using stereotypes. The macro declaration is just code written for example in the “Cpreprocessor” language and can be represented using standard KDM constructs, such as CodeElements, Action, Flow etc., if needed or light-weight extension elements, like Stereotypes and ExtendedValues. In many situations, the right implementation choice is to leave the directive as an empty container with a name, and likely, a SourceRef with a code snippet.

It is possible to provide a high-fidelity semantic-rich representation of the preprocessor directives themselves (for example, parameters to MacroUnit, the body of the MacroUnit as CodeElements, conditional compilation expressions as KDM micro actions and flows) however in most situations this is inadequate. Besides, since many preprocessors operate at the level of lexical tokens or below, the representation of the body of the MacroUnit may not be adequate at the level of the CodeModel. In many situations, only the resulting code in the primary language can be interpreted semantically and represented as meaningful KDM CodeElements. Relationship Alternative is a practical approximation of a more precise description of the flow between the alternative branches of the conditional compilation directive. Examples to this section only illustrate a simplified approach.

“Generated” code (code in the primary language that results from executing the preprocessor directives) is owned by a BlockUnit. There is a relation GeneratedFrom from the entire BlockUnit to the corresponding pre-processor directive. It is recommended that the generated code has a SourceRef with a snippet, since it is not present in the original source file and cannot be referred to using the SourceRegion construct.

“Included” code (code in the primary language that results from executing the pre-processor include directive) is also owned by a BlockUnit. There is a relation GeneratedFrom from the entire BlockUnit to the corresponding pre-processor directive. From the logical perspective, the contents of an included file are owned by a certain SharedUnit. The implementer may either clone the included code elements or to reuse them and keep a single copy in the SharedUnit. The recommended approach is to clone the elements of the SharedUnit.

The implementer of KDM has the following implementation choices:

- Ignore the embedded language constructs, represent original and generated primary code; the resulting KDM does not contain any pre-processor directives, and relationships expand and alternative; information about the embedded language cannot be recovered from the KDM representation.

- Ignore the generated primary code, provide a high-fidelity representation to the embedded construct; the embedded construct is the origin and the target of KDM relationships; some details of how the embedded construct is expanded into the primary code may not be recovered from the KDM instance (but in general, the embedded construct provides a better choice, since it is the construct introduced by the developer).
- Represent both the embedded constructs and the primary code, provide a high-fidelity representation only to the primary code; there is a BlockUnit that owns the generated code, the generated code is the origin and the target of KDM relationships; there are no KDM relations to and from the embedded construct (other than Expands and Alternative); there is a GeneratedFrom relation from the entire BlockUnit to the corresponding embedded construct.
- Represent both the embedded constructs and the primary code, provide the high fidelity representation to the embedded construct: there is a BlockUnit that owns the generated code, the embedded construct is the origin and the target of KDM relationships; there are no KDM relationships to and from the BlockUnit representing the generated code (but there may be some local relationships inside the BlockUnit); there is a GeneratedFrom relation from the entire BlockUnit to the corresponding embedded construct.

12.21.2 MacroUnit Class

MacroUnit class represents macro definitions common to several programming languages. Although KDM allows semantic-rich contents of MacroUnit (as it is a subclass of a ControlElement), usually it is sufficient to represent a macro definition only as a names KDM element that can be the target of special Expands relations, so that its usage in the primary code as well as in other macro definitions can be tracked. The kind attribute provides some additional semantic information about the macro definition.

Superclass

PreprocessorDirective

Attributes

kind:MacroKind	additional semantic properties of the macro definition

Semantics

MacroUnit represents a preprocessor directive that defines a named rule for generating code, usually in the form of a macrodefinition, possible with the parameters and the body, where the occurrence of the name of the macro with the actual parameters is substituted by the body of the macro definition. KDM does not explicitly represent parameters to the macro directive or the body of the macro definition, since the granularity of these objects is usually related to string manipulation. However, the optional owned code element may be used to represent these.

The implementer shall select a particular strategy to represent macro units.

12.21.3 MacroKind data type (enumeration)

MacroKind enumeration datatype describes several semantic classes of MacroUnits.

Literal Value

regular	Macro definition has a body and may have parameters.
option	Macro definition without a body and parameters, only a name.

undefined	This value represents an undefined macro as the target for some relations in the representation of default branches of conditional compilation and variants.
external	external compilation option
unknown	unknown class of a macro definition

12.21.4 MacroDirective Class

MacroDirective class represents the so-called “macro call,” the occurrence of a macro name (possible with parameters) in the primary code, such that the preprocessor recognizes it and “expands” by substituting the macro directive construct with its “definition.” A block of “generated” code elements that represent the primary code resulting from macro expansion may be associated with the MacroDirective.

Superclass

PreprocessorDirective

Semantics

MacroDirective represents the so-called “macrocall,” or an occurrence of a macro name (possibly with the actual parameters) which is substituted by the body of the macro definition in which the occurrences of the formal parameters are substituted by the corresponding actual parameters. A MacroDirective can own an optional action element as the origin of the action relationships to the actual parameters to the MacroDirective.

12.21.5 IncludeDirective Class

IncludeDirective class represents the so-called include directive, common to several programming languages and their preprocessors (for example, the COPY statement in Cobol, the #include directive in the C language). The include directive is usually related to a SharedUnit (for example, a copybook in Cobol, or a header file in C). A block of “included” code elements, which are the clones of the elements owned by the SharedUnit may be associated with the include directive. Semantics of the IncludeDirective class is described later in this section in more detail.

Superclass

PreprocessorDirective

Semantics

IncludeDirective represents a preprocessor directive that is related to copying the content of some SharedUnit into a stand-alone CompilationUnit.

12.21.6 Conditional Directive Class

ConditionalDirective class represents the so-called “variant” of a software system, resulting from the use of conditional compilation capabilities, common to several programming languages and their preprocessors (for example the #if … #endif and #ifdef … #endif directives of the preprocessor of the C language). ConditionalDirective represents a single “branch” of the conditional compilation construct. A block of “conditional” code elements that represent the elements of this particular variant that were selected for compilation may be associated with the conditional directive. Semantics of the ConditionalDirective class is described later in this section in more detail.

Superclass

PreprocessorDirective

Semantics

Conditional directive identifies a variant of the software system within a software product line that is controlled by the so-called conditional compilation. Conditional directive determines a block of “generated” code, corresponding to the selected variant. The block of “generated” code identifies the corresponding conditional directive.

12.22 PreprocessorRelations Class Diagram

The Preprocessor class diagram defines several concrete relationship types for the KDM support of embedded language constructs and pre-processor directives, common to several programming languages. The classes and associations of the PreprocessorRelations class diagram are shown at Figure 12.20.

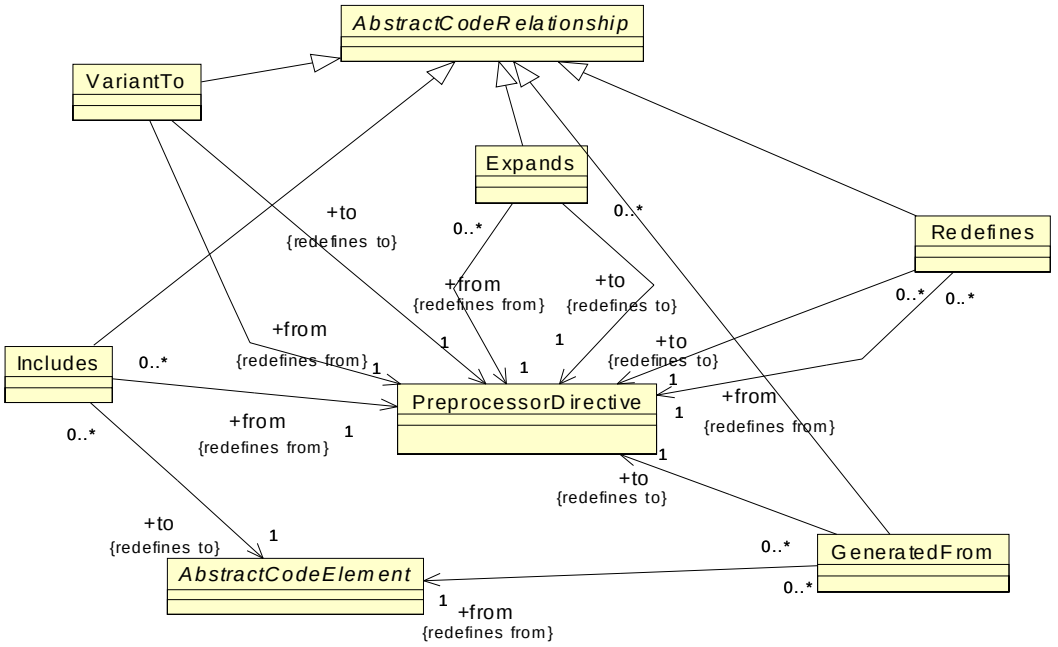

Figure 12.20 - PreprocessorRelations Class Diagram

12.22.1 Expands Class

Expands class represents the relationship between a MacroUnit to another MacroUnit or from a MacroDirective to a MacroUnit. This relationship results from using the name of the target macro definition in the context of the origin MacroUnit or MacroDirective.

Superclass

AbstractCodeRelationship

Associations

to:MacroUnit[1]	the target MacroUnit
from:PreprocessorDirective[1]	The origin context in which the MacroUnit is used.

Semantics

The implementer shall identify and represent associations between MacroUnits, as well as a MacroDirective and the corresponding MacroUnit according to the semantics of the preprocessor. See general description of the Preprocessor support for the implementer guidelines.

12.22.2 GeneratedFrom Class

GeneratedFrom class represents the relationship between a block of code elements that were not originally produced by the developers, but were produced by the preprocessor as the result of processing a certain preprocessor directive. In particular, according to the level of granularity selected in KDM, aligned with the concrete subclasses of the PreprocessorDirective class, the resulting code may represent one of the following:

- “generated” code that corresponds to a certain MacroDirective.
- “included” code that corresponds to a certain IncludeDirective.
- “conditional” code that was selected as a particular “variant” of a software product line with conditional compilation.

GeneratedFrom relationship originates from the entire BlockUnit that owns the “generated” code and target the corresponding PreprocessorDirective.

Superclass

AbstractCodeRelationship

Associations

to:PreprocessorDirective[1]	A subclass of a PreprocessorDirective class that represents the preprocessor directive that was involved in producing the code.
from:AbstractCodeElement[1]	The BlockUnit that owns the “generated” code.

Constraints

1. The origin of the GeneratedFrom relationship should be a BlockUnit.

Semantics

See the general description of the preprocessor directives for the implementer’s guidelines.

Example (C preprocessor)

```
#define GT(A,B) ((A) > (B))
#define GMAX(A,B)  g=( GT(A,B) ? (A) : (B))
GMAX(p+q, r+s );


<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
```

```
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Macro Directive Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CompilationUnit">
      <codeElement xmi:id="id.2" xmi:type="code:MacroUnit" name="GMAX">
        <source language="Cpreprocessor"
                snippet="#define GMAX(A,B) g=( GT(A,B) ? (A) : (B) )"/>
        <codeRelation xmi:id="id.3" xmi:type="code:Expands" to="id.4" from="id.2"/>
      </codeElement>
      <codeElement xmi:id="id.4" xmi:type="code:MacroUnit" name="GT">
        <source language="Cpreprocessor" snippet="#define GT(A,B) ((A) > (B))"/>
      </codeElement>
      <codeElement xmi:id="id.5" xmi:type="action:BlockUnit">
        <codeElement xmi:id="id.6" xmi:type="code:StorableUnit" name="p" type="id.49"/>
        <codeElement xmi:id="id.7" xmi:type="code:StorableUnit" name="q" type="id.49"/>
        <codeElement xmi:id="id.8" xmi:type="code:StorableUnit" name="r" type="id.49"/>
        <codeElement xmi:id="id.9" xmi:type="code:StorableUnit" name="s" type="id.49"/>
        <codeElement xmi:id="id.10" xmi:type="code:StorableUnit" name="g" type="id.49"/>
        <codeElement xmi:id="id.11" xmi:type="code:MacroDirective" name="m1">
          <source xmi:id="id.12" language="Cpreprocessor" snippet="GMAX(p+q,r+s);"/>
          <codeRelation xmi:id="id.13" xmi:type="code:Expands" to="id.2" from="id.11"/>
        </codeElement>
        <codeElement xmi:id="id.14" xmi:type="action:BlockUnit" name="bm1">
          <codeRelation xmi:id="id.15" xmi:type="code:GeneratedFrom" to="id.11" from="id.14"/>
          <codeElement xmi:id="id.16" xmi:type="action:ActionElement">
            <source xmi:id="id.17" language="C"
                    snippet="g=( ((p+q) > (r+s)) ? (p+q) : (r+s) );"/>
            <codeElement xmi:id="id.18" xmi:type="action:ActionElement" name="a1" kind="Add">
              <actionRelation xmi:id="id.19" xmi:type="action:Reads" to="id.6" from="id.18"/>
              <actionRelation xmi:id="id.20" xmi:type="action:Reads" to="id.10" from="id.18"/>
              <actionRelation xmi:id="id.21" xmi:type="action:Writes" to="id.47" from="id.18"/>
              <actionRelation xmi:id="id.22" xmi:type="action:Flow" to="id.23" from="id.18"/>
            </codeElement>
            <codeElement xmi:id="id.23" xmi:type="action:ActionElement" name="a2" kind="Add">
              <actionRelation xmi:id="id.24" xmi:type="action:Reads" to="id.8" from="id.23"/>
              <actionRelation xmi:id="id.25" xmi:type="action:Reads" to="id.9" from="id.23"/>
              <actionRelation xmi:id="id.26" xmi:type="action:Writes" to="id.48" from="id.23"/>
              <actionRelation xmi:id="id.27" xmi:type="action:Flow" from="id.23"/>
            </codeElement>
            <codeElement xmi:id="id.28" xmi:type="action:ActionElement"
                    name="a3" kind="GreaterThan">
              <codeElement xmi:id="id.29" xmi:type="code:StorableUnit"
                    name="c" type="id.50" kind="register"/>
              <actionRelation xmi:id="id.30" xmi:type="action:Reads" to="id.47" from="id.28"/>
              <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.48" from="id.28"/>
              <actionRelation xmi:id="id.32" xmi:type="action:Writes" to="id.29" from="id.28"/>
              <actionRelation xmi:id="id.33" xmi:type="action:Flow" to="id.34" from="id.28"/>
            </codeElement>
            <codeElement xmi:id="id.34" xmi:type="action:ActionElement"
                    name="a3.1" kind="Condition">
              <actionRelation xmi:id="id.35" xmi:type="action:Reads"
                    to="id.29" from="id.34"/>
              <actionRelation xmi:id="id.36" xmi:type="action:TrueFlow"
                    to="id.38" from="id.28"/>
              <actionRelation xmi:id="id.37" xmi:type="action:FalseFlow"
                    to="id.42" from="id.34"/>
            </codeElement>
            <codeElement xmi:id="id.38" xmi:type="action:ActionElement"
                    name="a4" kind="Assign">
              <actionRelation xmi:id="id.39" xmi:type="action:Reads" to="id.47" from="id.38"/>
              <actionRelation xmi:id="id.40" xmi:type="action:Writes" to="id.10" from="id.38"/>
              <actionRelation xmi:id="id.41" xmi:type="action:Flow" to="id.46" from="id.38"/>
            </codeElement>
            <codeElement xmi:id="id.42" xmi:type="action:ActionElement"
                    name="a5" kind="Assign">
              <actionRelation xmi:id="id.43" xmi:type="action:Reads" to="id.48" from="id.42"/>
              <actionRelation xmi:id="id.44" xmi:type="action:Writes" to="id.7" from="id.42"/>
              <actionRelation xmi:id="id.45" xmi:type="action:Flow" to="id.46" from="id.42"/>
            </codeElement>
            <codeElement xmi:id="id.46" xmi:type="action:ActionElement" name="a6" kind="Nop"/>
            <codeElement xmi:id="id.47" xmi:type="code:StorableUnit"
                    name="t1" type="id.49" kind="register"/>
            <codeElement xmi:id="id.48" xmi:type="code:StorableUnit"
                    name="t2" type="id.49" kind="register"/>
```

```
          </codeElement>
        </codeElement>
        <codeElement xmi:id="id.49" xmi:type="code:IntegerType" name="int"/>
        <codeElement xmi:id="id.50" xmi:type="code:BooleanType" name="boolean"/>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

12.22.3 Includes Class

Includes class represents the relationship from an IncludeDirective to a SharedUnit that represents the code elements being included.

Superclass

AbstractCodeRelationship

Associations

from:AbstractCodeElement[1]	the code elements being included (usually a SharedUnit)
from:PreprocessorDirective[1]	the IncludeDirective class that represents the include directive

Constraints

1. The origin of the Includes relationship should be an IncludeDirective.

Semantics

The implementer shall identify and represent include relationships according to the semantics of the particular preprocessor.

Example (C preprocessor)

```
/*---a.h---*/
... c1 ...
...c2...
/*---a.c---*/
#include "a.h"
...c1...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Include Directive Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <extensionFamily xmi:id="id.1" >
      <stereotype xmi:id="id.2" name="sample"/>
    </extensionFamily>
    <codeElement xmi:id="id.3" xmi:type="code:SharedUnit" name="a.h">
      <codeElement xmi:id="id.4" xmi:type="code:CodeElement" stereotype="id.2" name="c1">
        <source xmi:id="id.5" language="C"/>
      </codeElement>
      <codeElement xmi:id="id.6" xmi:type="code:CodeElement" stereotype="id.2" name="c2">
        <source xmi:id="id.7" language="C"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.8" xmi:type="code:CompilationUnit" name="a.c">
      <codeElement xmi:id="id.9" xmi:type="code:IncludeDirective">
```

```
        <source language="Cpreprocessor" snippet="#include &quot;a.h&quot;"/>
        <codeRelation xmi:id="id.10" xmi:type="code:Includes" to="id.3" from="id.9"/>
      </codeElement>
      <codeElement xmi:id="id.11" xmi:type="action:BlockUnit" name="b1">
        <codeRelation xmi:id="id.12" xmi:type="code:GeneratedFrom" to="id.9" from="id.11"/>
        <codeElement xmi:id="id.13" xmi:type="code:CodeElement"
             stereotype="id.2" name="c1_clone">
          <source xmi:id="id.14" language="C"/>
        </codeElement>
        <codeElement xmi:id="id.15" xmi:type="code:CodeElement"
             stereotype="id.2" name="c2_clone">
          <source xmi:id="id.16" language="C"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.17" xmi:type="action:BlockUnit" name="b2">
        <codeElement xmi:id="id.18" xmi:type="action:ActionElement" name="a1">
          <actionRelation xmi:id="id.19" xmi:type="action:ActionRelationship"
                  to="id.13" from="id.18"/>
        </codeElement>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

12.22.4 VariantTo Class

VariantTo class represents the relationship between variants of a software product line with conditional compilation. This relationship connects the ConditionalDirective to each alternative branch of the conditional compilation directive. KDM representation is expected to identify a single “default” variant, to which additional variants are alternatives. There is no VariantTo relationship to the “default” variant, only to the alternative ones. Each variant is expected to contain the relationship GeneratedFrom connecting it to the corresponding ConditionalDirective. The “default” variant is expected to have a VariantTo relationship to every alternative branch.

Superclass

AbstractCodeRelationship

Associations

to:PreprocessorDirective[1]	ConditionalDirective class that represents an alternative variant of the conditional.
from:PreprocessorDirective[1]	A ConditionalDirective class that represents the default variant of the conditional.

Constraints

1. The origin of the VariantTo relationship should be a ConditionalDirective.
2. The target of the VariantTo relationship should be a ConditionalDirective.

Semantics

The implementer shall identify and represent the variants and associations between the “generated” code and the corresponding conditional directive according to the semantics of the preprocessor. See the general description of the preprocessor directive support and the implementer guidelines.

Example (C preprocessor)

```
#define UNIX 1
#if UNIX | DEBUG
g=1;
```

```
#endif

Ifdef UNIX
g=1
#else
g=2
#endif


<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Variants Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:MacroUnit" name="UNIX">
      <source language="Cproprocessor" snippet="#define UNIX 1"/>
    </codeElement>
    <codeElement xmi:id="id.2" xmi:type="code:MacroUnit" name="DEBUG" kind="external"/>
    <codeElement xmi:id="id.3" xmi:type="code:StorableUnit" name="g" type="id.4">
      <codeElement xmi:id="id.4" xmi:type="code:IntegerType"/>
    </codeElement>
    <codeElement xmi:id="id.5" xmi:type="code:ConditionalDirective" name="c1">
      <source language="Cpreprocessor" snippet="#if UNIX | DEBUG"/>
      <codeRelation xmi:id="id.6" xmi:type="code:Expands" to="id.1" from="id.5"/>
      <codeRelation xmi:id="id.7" xmi:type="code:Expands" to="id.2" from="id.5"/>
    </codeElement>
    <codeElement xmi:id="id.8" xmi:type="action:BlockUnit" name="b1">
      <codeRelation xmi:id="id.9" xmi:type="code:GeneratedFrom" to="id.5" from="id.8"/>
      <codeElement xmi:id="id.10" xmi:type="action:ActionElement" name="a1" kind="Assign">
        <source xmi:id="id.11" language="C" snippet="g=123"/>
        <codeElement xmi:id="id.12" xmi:type="code:Value" name="123" type="id.4"/>
        <actionRelation xmi:id="id.13" xmi:type="action:Reads" to="id.12" from="id.10"/>
        <actionRelation xmi:id="id.14" xmi:type="action:Writes" to="id.3" from="id.10"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.15" xmi:type="code:ConditionalDirective" name="c2">
      <source language="Cpreprocessor" snippet="#ifdef UNIX"/>
      <codeRelation xmi:id="id.16" xmi:type="code:Expands" to="id.1" from="id.15"/>
      <codeRelation xmi:id="id.17" xmi:type="code:VariantTo" to="id.25" from="id.15"/>
    </codeElement>
    <codeElement xmi:id="id.18" xmi:type="action:BlockUnit" name="b2">
      <codeRelation xmi:id="id.19" xmi:type="code:GeneratedFrom" to="id.15" from="id.18"/>
      <codeElement xmi:id="id.20" xmi:type="action:ActionElement" name="a2" kind="Assign">
        <source xmi:id="id.21" language="C" snippet="g=123"/>
        <codeElement xmi:id="id.22" xmi:type="code:Value" name="1" type="id.4"/>
        <actionRelation xmi:id="id.23" xmi:type="action:Reads" to="id.22" from="id.20"/>
        <actionRelation xmi:id="id.24" xmi:type="action:Writes" to="id.3" from="id.20"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.25" xmi:type="code:ConditionalDirective" name="c3">
      <source language="Cpreprocessor" snippet="#else"/>
      <codeRelation xmi:id="id.26" xmi:type="code:Expands" to="id.1" from="id.25"/>
    </codeElement>
    <codeElement xmi:id="id.27" xmi:type="action:BlockUnit" name="b3">
      <codeRelation xmi:id="id.28" xmi:type="code:GeneratedFrom" to="id.25" from="id.27"/>
      <codeElement xmi:id="id.29" xmi:type="action:ActionElement" name="a3" kind="Assign">
        <source xmi:id="id.30" language="C" snippet="g=123"/>
        <codeElement xmi:id="id.31" xmi:type="code:Value" name="2" type="id.4"/>
        <actionRelation xmi:id="id.32" xmi:type="action:Reads" to="id.31" from="id.29"/>
        <actionRelation xmi:id="id.33" xmi:type="action:Writes" to="id.3" from="id.29"/>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

12.22.5 Redefines Class

Redefines class represents the relationship between a MacroUnit and another MacroUnit (usually with the same name) where the origin MacroUnit is a redefinition of the MacroUnit that is the target of the relationship. In many preprocessors, the redefinition is achieved simply by providing another macro definition with the same name. KDM Expands relationships are expected to correctly represent the semantics of the given preprocessor, by targeting the MacroUnit which is "current" definition at the given point.

Superclass

AbstractCodeRelationship

Associations

to:MacroUnit[1]	the old MacroUnit
from:PreprocessorDirective[1]	the new MacroUnit

Constraints

1. The origin of the Redefines relationship should be a MacroUnit.

Semantics

The implementer shall identify and represent redefinitions of macro units according to the semantics of the particular preprocessor.

Example (C preprocessor)

```
#define A 1
#define A 2
#undef A
#pragma once
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Preprocessor Directives example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <extensionFamily xmi:id="id.1" >
      <stereotype xmi:id="id.2" name="directive">
        <tag xmi:id="id.3" tag="directive_type" type="String"/>
      </stereotype>
    </extensionFamily>
    <codeElement xmi:id="id.4" xmi:type="code:MacroUnit" name="A">
      <source language="Cpreprocessor" snippet="#define A 1"/>
    </codeElement>
    <codeElement xmi:id="id.5" xmi:type="code:MacroUnit" name="DEBUG" kind="option">
      <source language="Cpreprocessor" snippet="#define DEBUG"/>
    </codeElement>
    <codeElement xmi:id="id.6" xmi:type="code:MacroUnit" name="A">
      <source language="Cpreprocessor" snippet="#define A 2"/>
      <codeRelation xmi:id="id.7" xmi:type="code:Redefines" to="id.4" from="id.6"/>
    </codeElement>
    <codeElement xmi:id="id.8" xmi:type="code:MacroUnit" name="A" kind="undefined">
      <source language="Cpreprocessor" snippet="#undef A"/>
      <codeRelation xmi:id="id.9" xmi:type="code:Redefines" to="id.6" from="id.8"/>
    </codeElement>
      <codeElement  xmi:id="id.10"  xmi:type="code:PreprocessorDirective"  stereotype="id.2"
name="d1">
      <taggedValue xmi:id="id.11" xmi:type="kdm:TaggedValue" tag="id.3" value="pragma once"/>
```

```
      <source language="Cpreprocessor" snippet="#pragma once"/>
    </codeElement>
  </model>
</kdm:Segment>
```

Section V - Miscellaneous Code Elements

12.23 Comments Class Diagram

The Comments class diagram defines meta-model elements that represent comments. Comment is at the bottom of the inheritance hierarchy so that it can occur in all containers without restrictions. The classes and associations of the Comments diagram are shown in Figure 12.21.

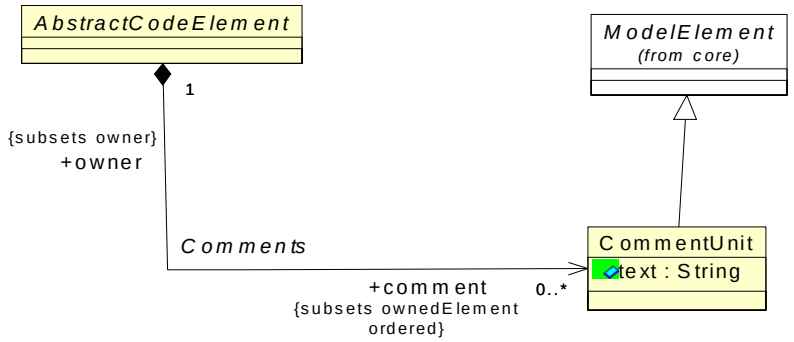

Figure 12.21 - Comments Class Diagram

12.23.1 CommentUnit Class

The CommentUnit is a meta-model element that represents comments in existing systems (including any special comments). CommentUnit element can be used to introduce comments during transformation of the existing system (including special comments).

Superclass

ModelElement

Attributes

text:String	The representation of the comment.

Semantics

CommentUnit represents comments in the source code. CommentUnits are associated with a certain code element. The implementer shall make an adequate decision on how to associate line comments with the surrounding elements in the source code.

12.23.2 AbstractCodeElement Class (additional properties)

Associations

comment:CommentUnit[0..*]	CommentUnits associated with the AbstractCodeElement

Semantics

12.24 Visibility Class Diagram

The Visibility class diagram defines meta-model elements that represent visibility of code elements in their corresponding containers. The classes and associations that make up the Visibility diagram are shown in Figure 12.22.

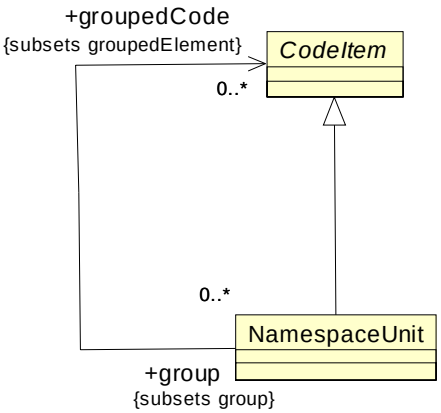

Figure 12.22 - Visibility Class Diagram

12.24.1 Namespace Class

The Namespace is a specific meta-model element that represents the target of the VisibleIn or Imports visibility relationships.

Superclass

CodeItem

Associations

groupedCode:CodeItem[0..*]	A KDM group of code elements that belong to the namespace. The actual owners of these elements are the corresponding modules, not the namespace, since namespaces can, in general cross cut the module boundaries.

Constraints

1. Namespace element should not belong to own group.

Semantics

A Namespace is a group of code elements. A Namespace can be owned by Module element or one of its subclasses. Namespace class represents a unit of visibility (for example, the namespace concept in C++).

An anonymous namespace can represent a group of code elements that are the target of an Imports relationship.

12.25 VisibilityRelations Class Diagram

The VisibilityRelations class diagram defines meta-model elements that represent visibility of code elements in their corresponding containers. The classes and associations of the Visibility diagram are shown in Figure 12.23.

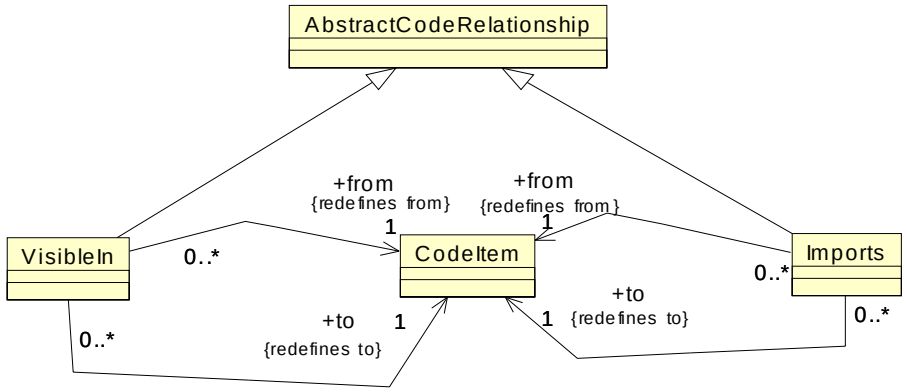

Figure 12.23 - VisibilityRelations Class Diagram

12.25.1 VisibleIn Class

The VisibleIn is a specific meta-model element that represents semantic relation between two code items, where one provides the restricted visibility context for another code item.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	The CodeItem visibility of which is specified.
to:CodeItem[1]	The CodeItem that provides the visibility context.

Semantics

VisibleIn optional relationship represents an association between a code item and one of the containers that corresponds to the visibility scope of the first item. This relationship is optional, since all other KDM relationships are determined by the semantics of the target language, including the visibility rules.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
name="Visibility and Comment Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CodeAssembly">
      <codeElement xmi:id="id.2" xmi:type="code:NamespaceUnit"
                   name="ab" groupedCode="id.4 id.9 id.13"/>
      <codeElement xmi:id="id.3" xmi:type="code:CompilationUnit" name="a">
        <codeElement xmi:id="id.4" xmi:type="code:CallableUnit"
                     name="foo" type="id.8" kind="regular">
          <comment text="Comment #1 to foo"/>
          <comment text="Comment #2 to foo"/>
          <codeRelation xmi:id="id.5" xmi:type="code:VisibleIn" to="id.2" from="id.4"/>
          <codeElement xmi:id="id.6" xmi:type="action:ActionElement" name="a1">
            <comment xmi:id="id.7" text="Comment to action element a1"/>
          </codeElement>
          <codeElement xmi:id="id.8" xmi:type="code:Signature" name="foo"/>
        </codeElement>
        <codeElement xmi:id="id.9" xmi:type="code:IntegerType" name="int">
          <comment xmi:id="id.10" text="Comment to integer type"/>
          <codeRelation xmi:id="id.11" xmi:type="code:VisibleIn" to="id.2"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.12" xmi:type="code:CompilationUnit" name="b">
        <codeElement xmi:id="id.13" xmi:type="code:RecordType" name="bar">
          <comment xmi:id="id.14" text="Comment to record type bar"/>
          <codeRelation xmi:id="id.15" xmi:type="code:VisibleIn" to="id.2" from="id.13"/>
          <itemUnit xmi:id="id.16" name="foobar" type="id.9">
            <comment xmi:id="id.17" text="Comment to item unit foobar"/>
            <codeRelation xmi:id="id.18" xmi:type="code:VisibleIn" to="id.13" from="id.16"/>
          </itemUnit>
        </codeElement>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

12.25.2 Imports Class

The Imports meta-model element represents an association between two CodeItems where one CodeItem “imports” definitions from another. The “import” relationship is common to several programming languages (for example, the import statement in Java). In this relationship the origin CodeItem (usually, a CompilationUnit or a subclass of Module) resolves the visibility of certain names that are defined (owned) by the target CodeItem (usually, another CompilationUnit or some other subclass of Module, but possibly a NamespaceUnit from another CodeItem, or even an individual code element). The Imports class simply represents the “import” relationships between CodeItem, for example, for tracking dependencies between packages. KDM representations themselves do not require additional import statements in order to have relationships between CodeItem, or even between different models.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	A subclass of CodeResource that represents the “consumer” of the imported definitions.
to:CodeItem[1]	A subclass of CodeResource that represents the “owner” of the imported definitions.

Constraints

1. The origin of the Imports relationship should be a subclass of Module.

Semantics

The implementer shall identify and represent import directives and their targets according to the semantics of the programming language of the existing software system.

12.26 ExtendedCodeElements Class Diagram

The ExtendedCodeElements class diagram defines two “wildcard” generic elements for the code model as determined by the KDM model pattern: a generic code entity and a generic code relationship.

The classes and associations of the ExtendedCodeElements diagram are shown in Figure 12.24.

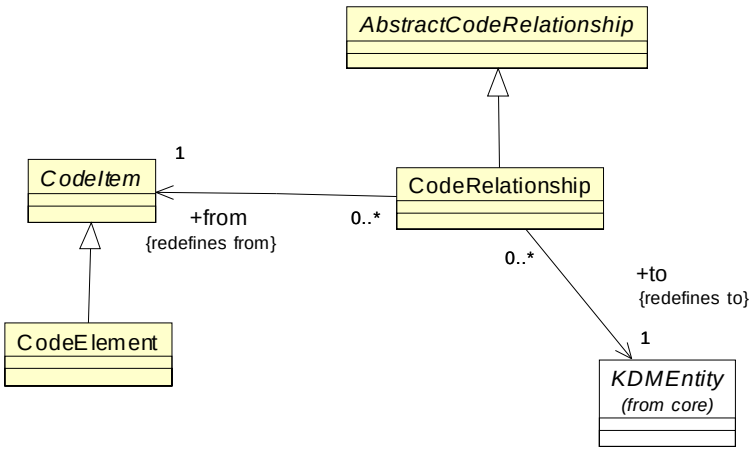

Figure 12.24 - ExtendedCodeElements Class Diagram

12.26.1 CodeElement Class (generic)

The CodeElement is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

CodeItem

Constraints

1. CodeElement should have at least one stereotype.

Semantics

A code entity with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity type of the code model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

12.26.2 CodeRelationship Class (generic)

The CodeRelationship is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractCodeRelationship

Associations

from:CodeItem[1]	the CodeItem
to:KDMEntity[1]	the KDMEntity

Constraints

1. CodeRelationship should have at least one stereotype.

Semantics

A code relationship with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model relationship type of the code model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

13 Action Package

13.1 Overview

The Action package defines a set of meta-model elements whose purpose is to represent implementation-level behavior descriptions determined by programming languages, for example statements, operators, conditions, features, as well as their associations, for example control and data flow. Action package extends the KDM Code package. As a general rule, in a given KDM instance, each instance of an action element represents some programming language construct, determined by the programming language of the existing software system. Each instance of an action meta-model element corresponds to a certain region of the source code in one of the artifacts of the software system. An action element usually represents one or more statements, and an action relationship usually represents a usage of a name in a statement.

13.2 Organization of the Action Package

The Action package consists of the following 11 class diagrams.

1. ActionElements
2. ActionFlow
3. ActionInheritances
4. CallableRelations
5. DataRelations
6. ExceptionBlocks
7. ExceptionFlow
8. ExceptionRelations
9. InterfaceRelations
10. UsesRelations
11. ExtendedActionElements

The Action package depends on the following packages:

- Core
- kdm
- Source
- Core

13.3 ActionElements Class Diagram

In general, the Action package follows the uniform pattern for KDM models and extends the KDM Framework with specific meta-model elements related to implementation-level behavior. The Action package deviates from a uniform pattern for KDM models because the Action package does not define a separate KDM model, but rather extends the Code

model, defined in the Code package. Therefore each Action element is a subclass of AbstractCodeElement. Action package defines most of the relationship types to the Code model. Together, Action and Code packages constitute the Program Elements Layer of KDM.

The ActionElements diagram defines the following classes determined by the KDM model pattern:

- ActionElement – main class of the Action package.
- AbstractActionRelationship - a class representing an abstract parent of all KDM relationships that can be used to represent actions (in general, related to usages of names in programming language statements).

The class diagram shown in Figure 13.1 captures these classes and their relations.

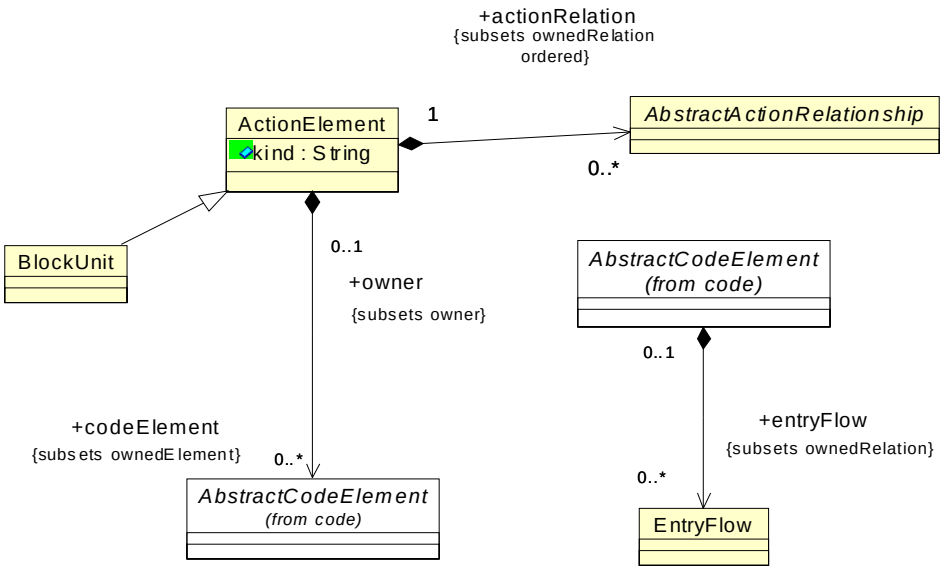

Figure 13.1 - ActionElements Class Diagram

13.3.1 ActionElement Class

The ActionElement is a class to describe a basic unit of behavior. ActionElements are endpoints for primitive relations. ActionElement can be linked to the original representation through the SourceRef element from the Source package.

Superclass

AbstractCodeElement

Attributes

kind:String	Represents the meaning of the operations, performed by the ActionElement.

Associations

actionRelation:ActionRelationship[0..*]	Action relationships originating from the given action element.
codeElement: AbstractCodeElement[0..*]	Owned code elements (for example, nested action elements, or nested BlockUnits, or nested definitions of datatypes and computational objects).

Constraints

1. ActionElement may own StorableUnit or ValueElement and their subclasses and nested action elements.

Semantics

An action element represents a unit of behavior. It can represent one or more programming language statements, or even a part of a programming language statement. The implementer shall select the granularity of the action elements. As a minimum, each ControlElement should own at least one ActionElement so that it can be the endpoint of all ActionRelationships originating from the corresponding ControlElement. Static analysis grade KDM implementations should use well-defined fine grained ActionElements, specified as the “micro KDM” compliance point.

Data elements owned by the ActionElement represent temporary variables or values used by the action element. Action elements may be nested, when one ActionElement instance is a container for other ActionElements.

For micro KDM compliance the list of the allowed action element kind values and their meaning is described in Annex A. For non-micro KDM implementation, the value of the kind is not normative.

The implementer shall map programming language statements and other descriptions of behavior into KDM ActionElements.

An ActionElement can own other ActionElements. Such ActionElement is called a “composite action.” Composite action represents an entire set of leaf actions owned directly or indirectly.

13.3.2 AbstractActionRelationship Class (abstract)

The AbstractActionRelationship is the parent class representing various KDM relationships that originate from an ActionElement.

Superclass

KDMRelationship

Semantics

Usually, an action relationship corresponds to some usage of a name in a programming language statement. Action relationships originate from ActionElements as opposed to code relationships that originate from code elements. AbstractActionRelationship is subclassed by several concrete action relationship meta-elements.

13.3.3 BlockUnit Class

The BlockUnit represents logically and physically related blocks of ActionElement, for example, blocks of statements.

Superclass

ActionElement

Associations

codeElement:AbstractCodeElement[0..*]	owned code elements including nested BlockUnits

Semantics

A BlockUnit is a logical container for action elements. BlockUnit is similar to a composite ActionElement that can also contain nested ActionElement and data elements. BlockUnit represents the entire set of leaf Actions, owned by the BlockUnit directly or indirectly.

13.3.4 AbstractCodeElement (additional properties)

Associations

entryFlow:EntryFlow[0..*]	EntryFlow relationships that associate the given abstract code element and some action elements owned by it, which are designated as the entry actions.

Semantics

Control flow is transferred to the Entry actions when the AbstractCodeElement that owns them is entered. Multiple EntryFlow elements represent nondeterministic control flow.

13.4 ActionInheritances Class Diagram

The ActionInheritances class diagram is determined by the uniform pattern for extending the KDM framework. Action package does not define a separate KDM model, but extends the Code model. The class diagram shown in Figure 13.2 captures these classes and their associations.

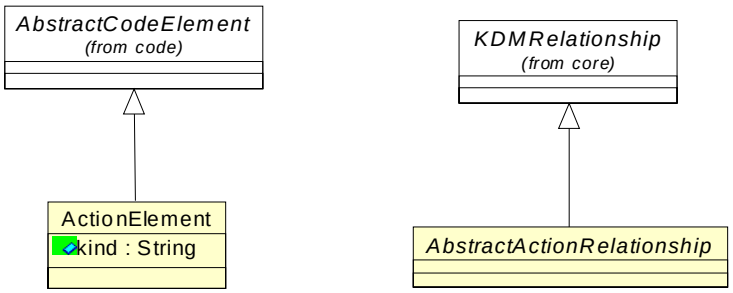

Figure 13.2 - ActionInheritances Class Diagram

13.5 ActionFlow Class Diagram

The ActionFlow class diagram provides basic meta-model constructs to define the control-flow between ActionElements. The class diagram shown in Figure 13.3 captures these classes and their relations.

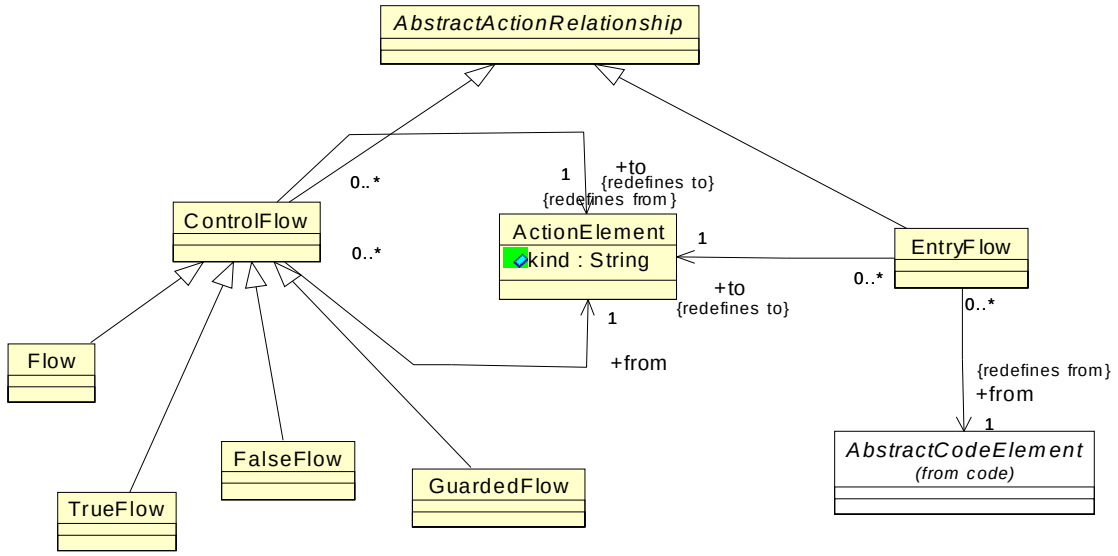

Figure 13.3 - ActionFlow Class Diagram

13.5.1 ControlFlow Class (generic)

The ControlFlow is a generic modeling element that represents control flow relation between two ActionElements. It is further subclassed with more specific modeling elements. ControlFlow class could be used to represent programming constructs that are not covered by any of the specific subclasses as an *extension point*.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	the origin of the control flow
to:ActionElement[1]	The target of control flow, when represented by the next action element in the trace determined by the control flow.

Constraints

1. ControlFlow class should always be used with a stereotype.

Semantics

From the KDM representation perspective, ControlFlow meta-element is an extension point that can be used as the base element for new “virtual” meta-model elements, representing specific control flow relationships not covered by the semantic categories of its concrete subclasses. From the meta-model perspective, the ControlFlow element defines the common properties of various control flow relationship representations in KDM.

Multiple ControlFlow elements represent nondeterministic choice of behavior.

The implementer shall map control flow mechanisms of the given programming language into ControlFlow meta-elements. The implementer shall adequately represent the control flows of the existing system by a set of action elements and ControlFlow relationships between them.

13.5.2 EntryFlow Class

The EntryFlow is a modeling element that represents an initial flow of control into a KDM element. The EntryFlow relationship is used in a uniform way for describing entry points to other KDM code elements. It should be used to represent any type of special entry flows, such as the entry from a CodeAssembly to the initialization code block or action, from Module to initialization block, from a callable unit to the inititialization block, from a class to the initialization block or from a compound action to the first internal action.

Superclass

AbstractActionRelationship

Associations

from:AbstractCodeElement[1]	AbstractCodeElement that owns one or more action elements that represents its behavior.
to:ActionElement[1]	The action element that is selected when the owner element is called.

Constraints

1. Each AbstractCodeElement or its subclass such that it owns one or more ActionElement should have a corresponding EntryFlow relationship in which the “from” attribute is the AbstractCodeElement.
2. The “to” attribute of an EntryFlow element should be an ActionElement that is owned by the AbstractCodeElement that is the “from” attribute of the EntryFlow.

Semantics

The “entry” action element is the first action element in a trace that corresponds to the behavior of the owner AbstractCodeElement. Multiple action elements that are designated as “entry” actions, represent a nondeterministic choice, i.e., several possible trace families, each of which start with a different “entry” action element.

1. Represent initialization code as BlockUnit element with special micro action kind “Init.”
2. Initialization code can belong to a ControlElement, CompilationUnit, or CodeAssembly.
3. The EntryFlow relationship is used in a uniform way for describing entry points to other KDM code elements. It should be used for any type of special flows, e.g., entry to a CodeAssembly to init Block or action, from Module to init block, from callable unit to init block, from class to init block, or from compound action to the first internal action.
4. The CodeAssembly should have custom initialization block that consists of a sequence of action elements, including action elements with action kind=”Init” and an EntryFlow relation to the initialization blocks of the owned CompilationUnits (and other owned elements when appropriate), and an action element with action kind=”Calls” and a Calls relation to the logical entry point (for example, the CallableUnit “main”). The initialziation blocks of compilation units referred to by custom initialization block in a CodeAssembly do not need

to have the Flow relationship at their respective last action element. The control flow is resumed with the Flow relationship of the initialization action in the custom initialization block. See example at "HasValue Class" on page 110.

13.5.3 Flow Class

The Flow class is a modeling element that represents control flow relationship between two ActionElements such that the ActionElement that corresponds to the "to" attribute of the Flow is a successor of the ActionElement that corresponds to the "from" attribute of the Flow.

Superclass

ControlFlow

Constraints

1. If there is one or more Flow elements there should be no TrueFlow, FalseFlow, or GuardedFlow with the same action element as the "from" attribute.

Semantics

If there exists two or more Flow elements, such that they share the same ActionElement as the "from" attribute, they represent an unspecified flow of control.

13.5.4 TrueFlow Class

The TrueFlow class is a modeling element that represents control flow relationship between two ActionElements such that

- the ActionElement that corresponds to the "from" attribute of the TrueFlow represents the logical condition; and
- the ActionElement that corresponds to the "to" attribute of the TrueFlow is a successor of the ActionElement that corresponds to the "from" attribute of the TrueFlow when the value of the condition is true.

Superclass

ControlFlow

Constraints

1. If there exists an ActionElement with a TrueFlow element, there should be no GuardedFlow or Flow elements that have the same ActionElement as the "from" attribute (but there can be FalseFlow).

Semantics

If there exists two or more TrueFlow elements, such that they share the same ActionElement as the "from" attribute, they represent an unspecified flow of control. If there is no FalseFlow element, then the current ActionElement is terminal, when condition is not satisfied.

13.5.5 FalseFlow Class

The FalseFlow class is a modeling element that represents control flow relationship between two ActionElements such that

- the ActionElement that corresponds to the "from" attribute of the FalseFlow represents the logical condition, and

- the ActionElement that corresponds to the “to” attribute of the FalseFlow is a successor of the ActionElement that corresponds to the “from” attribute of the FalseFlow when the value of other conditions is not satisfied.

FalseFlow represents the “default” control flow branch in representing conditional statements and switch statements. FalseFlow is a specific modeling element that is used in combination with either TrueFlow or GuardedFlow.

Superclass

ControlFlow

Constraints

1. If there exists a FalseFlow element, there should be either:

    - a corresponding TrueFlow element such that both the TrueFlow and the FalseFlow elements have the same ActionElement as the “from” attribute, or

    - one or more GuardedFlow elements that have the same ActionElement as the “from” attribute, and there are no other relationship elements that are subclasses of FlowRelationship that have the same ActionElement as the “from” attribute.

Semantics

If there exists two or more FalseFlow elements, such that they share the same ActionElement as the “from” attribute, they represent an unspecified flow of control.

13.5.6 GuardedFlow Class

The GuardedFlow class is a modeling element that represents control flow relationship between two or more ActionElements such that:

- the ActionElement that corresponds to the “from” attribute of the GuardedFlow represents the selection statement (for example, a “switch” statement); and

- the ActionElement that corresponds to the “to” attribute of the GuardedFlow represents the guarding condition that determines a branch of control flow; and

- the branch of control flow determined by the ActionElement that corresponds to the “to” attribute of the GuardedFlow element is a successor of the ActionElement that corresponds to the “from” attribute of the GuardedFlow when the guarded condition in the context of the selection statement is satisfied.

FalseFlow element can be used in combination with one or more GuardedFlow elements to represent the default branch of control flow, for example, to represent the default branch of a switch statement.

Superclass

ControlFlow

Constraints

1. If there exists a GuardedFlow element, there should be no other relationship elements that are subclasses of FlowRelationship that have the same ActionElement as the “from” attribute, with exception of one or more GuardedFlow elements or zero or one FalseFlow element.

Semantics

In micro-KDM conformant implementations the ActionElement that corresponds to the “to” attribute of the GuardedFlow has kind=”Guard.” It has a Reads relationship to the value of the guard for the corresponding branch.

13.6 CallableRelations Class Diagram

The CallableRelations class diagram defines a set of meta-model elements to represent call-type behavior that associate ActionElement to ControlElement and represent control flows of the existing software system.

The CallableRelations diagram describes the following types:

- Calls - is a modeling element that represents a call-type relationship between an ActionElement and a CallableElement or one of its subclass elements, in which the ActionElement represents some form of a call statement, and the CallableElement represents the element being called.
- Dispatches - is a modeling element that represents a call-type of relationship between an ActionElement and a data item, in which the ActionElement represents some form of a call, and the data item represents a pointer to a procedure type.

The class diagram shown in Figure 13.4 captures these classes and their relations.

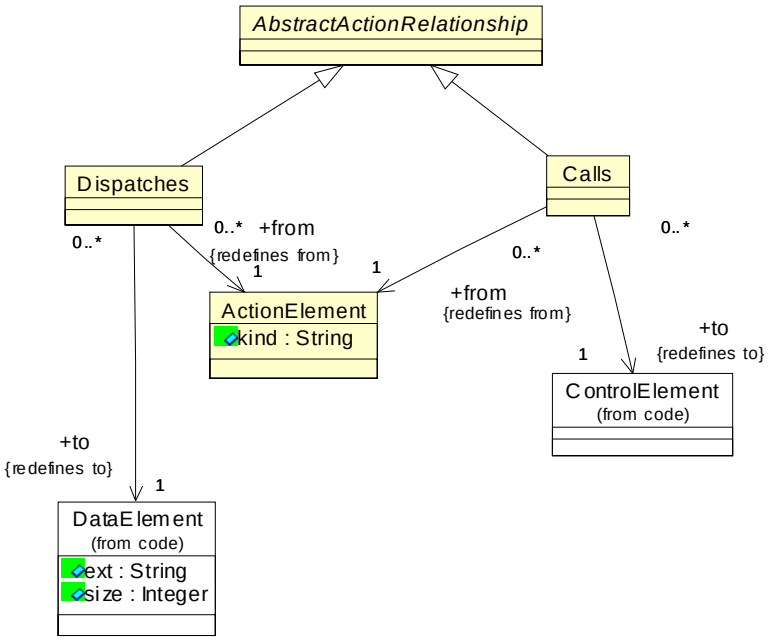

Figure 13.4 - CallableRelations Class Diagram

13.6.1 Calls Class

Calls is a modeling element that represents a call-type relationship between an ActionElement and a ControlElement or one of its subclass elements. The ActionElement represents some form of a call statement, and the ControlElement represents the element being called. In the meta-model the Calls element is a subclass of ActionRelationship.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	the action element from which the call relation originates
to:ControlElement[1]	the target ControlElement

Semantics

Calls relationship corresponds to the ISO/IEC 11404 “invoke” operation on a procedure type. It can represent a call to a procedure, a static method, a non-static method of a particular object instance, a virtual method, or an interface element.

Calls relation to a non-static method should be accompanied by an Addresses relationship to the corresponding object.

Precise semantics of a call can be represented by the “kind” element of the owner ActionElement, according to the guidelines provided in the “micro KDM” compliance point.

13.6.2 Dispatches Class

Dispatches is a modeling element that represents a call-type of relationship between an ActionElement and a data item. The ActionElement represents some form of a call behavior, and the data item represents a pointer to a procedure type.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element from which the call relation originates.
to:DataElement[1]	The data element that represents the pointer to a procedure type.

Semantics

Dispatches relation does not identify the actual target of the call. Additional analysis of the KDM instance may be required to determine the real targets of the Dispatches call.

Example (C)

```
typedef int(*fp)(int i );
int foo(int i){}
int bar( int i) {}
void foobar() {
       fp pf;
       pf=foo;
       pf=bar;
       *pf(1);
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
```

```
      xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
      xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="Dispatch Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CompilationUnit" name="Dispatch.c">
      <codeElement xmi:id="id.2" xmi:type="code:CallableUnit"
                   name="foo" type="id.15" kind="regular">
        <codeRelation xmi:id="id.3" xmi:type="code:HasType" to="id.14" from="id.2"/>
        <codeElement xmi:id="id.4" xmi:type="code:Signature" name="foo">
          <parameterUnit xmi:id="id.5" name="a" type="id.13"/>
          <parameterUnit xmi:id="id.6" type="id.13" kind="return"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.7" xmi:type="code:CallableUnit"
                   name="bar" type="id.15" kind="regular">
        <codeRelation xmi:id="id.8" xmi:type="code:HasType" to="id.14" from="id.7"/>
        <codeElement xmi:id="id.9" xmi:type="code:Signature" name="bar">
          <parameterUnit xmi:id="id.10" name="a" type="id.13"/>
          <parameterUnit xmi:id="id.11" type="id.13" kind="return"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.12" xmi:type="code:StorableUnit" name="pf" type="id.14"/>
      <codeElement xmi:id="id.13" xmi:type="code:IntegerType" name="int"/>
      <codeElement xmi:id="id.14" xmi:type="code:TypeUnit" name="fp" type="id.15">
        <codeElement xmi:id="id.15" xmi:type="code:Signature" name="f">
          <parameterUnit xmi:id="id.16" name="a" type="id.13"/>
          <parameterUnit xmi:id="id.17" type="id.13" kind="return"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.18" xmi:type="code:CallableUnit" name="foobar" type="id.33">
        <entryFlow xmi:id="id.19" to="id.20" from="id.18"/>
        <codeElement xmi:id="id.20" xmi:type="action:ActionElement" name="a1" kind="Assign">
          <actionRelation xmi:id="id.21" xmi:type="action:Addresses" to="id.2" from="id.20"/>
          <actionRelation xmi:id="id.22" xmi:type="action:Writes" to="id.12" from="id.20"/>
          <actionRelation xmi:id="id.23" xmi:type="action:Flow" to="id.24" from="id.20"/>
        </codeElement>
        <codeElement xmi:id="id.24" xmi:type="action:ActionElement" name="a2" kind="Assign">
          <actionRelation xmi:id="id.25" xmi:type="action:Addresses" to="id.2" from="id.24"/>
          <actionRelation xmi:id="id.26" xmi:type="action:Writes" to="id.12" from="id.24"/>
          <actionRelation xmi:id="id.27" xmi:type="action:Flow" to="id.28" from="id.24"/>
        </codeElement>
        <codeElement xmi:id="id.28" xmi:type="action:ActionElement" name="a3" kind="PtrCall">
          <codeElement xmi:id="id.29" xmi:type="code:Value" name="1" type="id.13"/>
          <actionRelation xmi:id="id.30" xmi:type="action:Reads" to="id.12" from="id.28"/>
          <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.29" from="id.28"/>
          <actionRelation xmi:id="id.32" xmi:type="action:Dispatches"
                   to="id.12" from="id.28"/>
        </codeElement>
        <codeElement xmi:id="id.33" xmi:type="code:Signature" name="foobar"/>
      </codeElement>
    </codeElement>
  </model>
</kdm:Segment>
```

13.7 DataRelations Class Diagram

The DataRelations class diagram defines a set of meta-model elements that represent data flow relationships between action elements and data elements. The “from” endpoint of these relationships is some ActionElement that represents a statement that involves a data operation. The “to” endpoint represents some code item. Data relationships are shown at Figure 13.5.

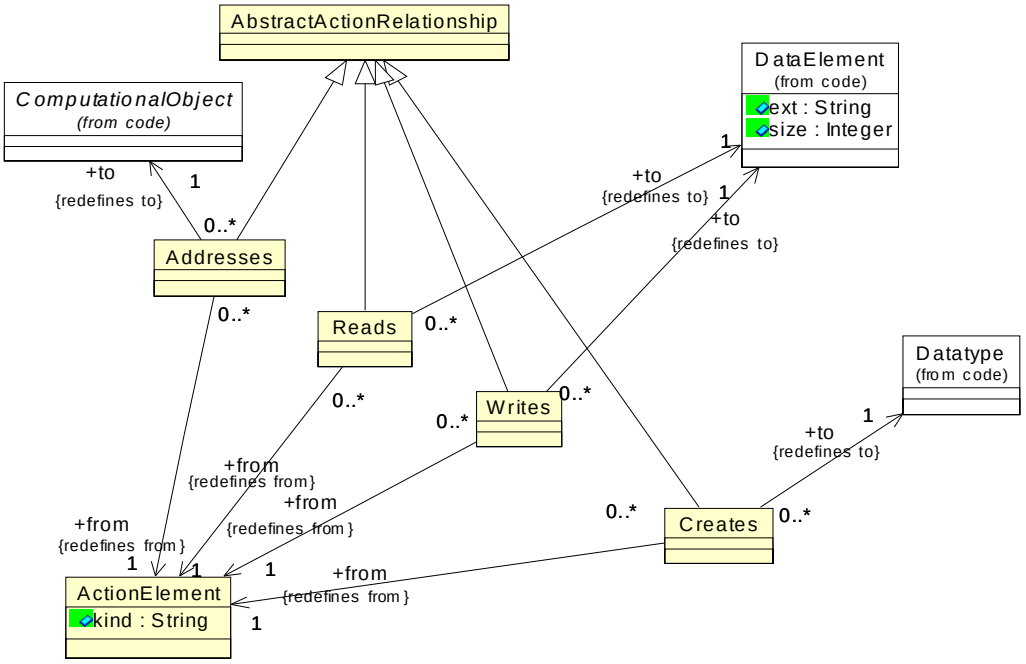

Figure 13.5 - DataRelations Class Diagram

13.7.1 Reads Class

The Reads relationship represents a flow of data from a DataElement to an ActionElement (read access to the DataElement).

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element that owns the Reads relationship.
to:DataElement[1]	The DataElement that is the source of the flow of data.

Semantics

Reads relationship represents an association between an action element, which implements a flow of data **from** a certain data element to the corresponding data element according to the semantics of the programming language of the existing software system.

13.7.2 Writes Class

The Writes represents flow of data from an ActionElement to a DataElement (write access to the DataElement).

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element that owns the Writes relationship.
to:DataElement[1]	The DataElement that is the sink of the flow of data.

Semantics

Writes relationship represents an association between an action element, which implements a flow of data **to** a certain data element to the corresponding data element according to the semantics of the programming language of the existing software system.

13.7.3 Addresses Class

Addresses relationship represents access to a complex data structure, where the Reads or Writes relationships are applied to the elements of the data structure, or where an address of a data element is taken.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element that owns the Addresses relationship.
to:ComputationalObject[1]	The Computational object that is being accessed.

Semantics

Addresses relationship represents an association between an action element that receives a reference to a certain data element to the corresponding data element according to the semantics of the programming language of the existing software system.

13.7.4 Creates Class

The Creates represents an association between an ActionElement and a Datatype such that the ActionElement creates a new instance of the Datatype.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element that owns the Creates relationship.
to:Datatype[1]	The DataElement that is instantiated by the ActionElement.

Semantics

Creates relationship represents an association between an action element that creates a new instance of a certain data element to the corresponding datatype according to the semantics of the programming language of the existing software system.

13.8 ExceptionBlocks Class Diagram

The Exceptions class diagram defines meta-model elements that represent containers involved in exception-handling mechanism common to several programming languages. These classes are illustrated at Figure 13.6.

Basic try-, catch- and finally-blocks are represented by dedicated containers defined as subclasses of the ExceptionUnit to represent the contents of those blocks. The TryUnit provides the containment to track the dependent Catch clauses and the optional finally block (for languages that support it).

The Catch container can own ParameterUnit to represent the data passed to it by the exception-handling mechanism when this block is activated, the so-called “its catch object.” This allows exceptions to be modeled like any other object and to have their own inheritance. The ParameterUnit that represents the “catch object” of a catch-block has special ParameterKind value kind=”exception” to represent parameter passing via exception mechanism or kind=”catchall” to represent the catch all construct in C++.

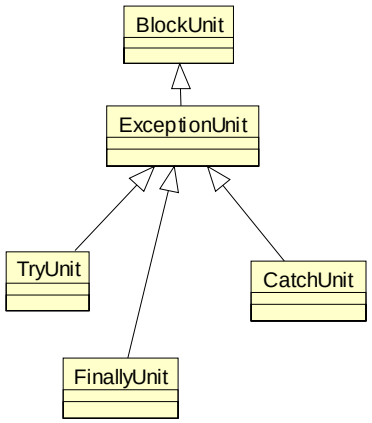

Figure 13.6 - ExceptionBlocks Class Diagram

13.8.1 ExceptionUnit Class

ExceptionUnit class is a generic meta-model element that provides a common superclass to various KDM containers for representing exception handling. ExceptionUnit is a subclass of a BlockUnit. ExceptionUnit is a KDM container, which can own both ActionElement (for example, statements in the catch-block) as well as CodeItem (for example, parameters to the catch-block, local definitions, and nested blocks). ExceptionUnit is a generic element, and KDM models are expected to use concrete subclasses of ExceptionUnit with more precise semantics. However, ExceptionUnit can be used as an extended modeling element with a stereotype. ExceptionUnit is more specific than a BlockUnit.

Superclass

BlockUnit

Constraints

1. ExceptionUnit should have a stereotype.

Semantics

13.8.2 TryUnit Class

TryUnit class is a meta-model element that represents try-blocks common to several programming languages. TryUnit is a container for action elements and associated definitions of CodeItems. The purpose of a TryUnit is to represent try-blocks and to manage exception flow to related catch-blocks and finally-block (for programming languages that support this concept). In particular, the TryUnit is the origin of ExceptionFlow relations to the corresponding CatchUnits and FinallyUnit blocks, which represent related catch- and finally-blocks. TryUnit may own nested TryUnit blocks.

Superclass

ExceptionUnit

Semantics

TryUnit represents a try-block.

13.8.3 CatchUnit Class

CatchUnit is a meta-model element that represents catch-blocks. Particular CatchUnit should be associated to a particular TryUnit through ExceptionFlow relation. CatchUnit may contain ParameterUnit that represents the exception object passed to the catch-block by the exception-handling mechanism.

Superclass

ExceptionUnit

Constraints

1. CatchUnit should be associated to a TryUnit. In particular, a CatchUnit should be the target of an ExceptionFlow relationship that originates from some TryUnit.

Semantics

CatchUnit represents one of the catch-blocks associated with a certain try-block. Usually, one or more CatchUnits follow a TryUnit. Each CatchUnit should be associated with the corresponding TryUnit through an instance of ExceptionFlow relationship.

13.8.4 FinallyUnit Class

FinallyUnit is a meta-model element that represents finally-block associated with a certain try-block. The FinallyUnit is associated with the core responding TryUnit through an ExitFlow relation.

Superclass

ExceptionUnit

Constraints

1. FinallyBlock should be associated to a TryUnit. In particular, a FinallyUnit should be the target of an ExitFlow relationship that originates from some TryUnit.

2. TryUnit should not be associated with more than one FinallyBlock.

Semantics

FinallyBlock represents a finally-block associated with a certain try-block.

Example

```
.<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="Exceptions Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:ClassUnit" name="A">
      <codeElement xmi:id="id.2" xmi:type="code:MethodUnit" name="foo">
        <entryFlow xmi:id="id.3" to="id.4" from="id.2"/>
        <codeElement xmi:id="id.4" xmi:type="action:TryUnit" name="t1">
          <codeElement xmi:id="id.5" xmi:type="action:ActionElement" name="a1" kind="Call">
            <actionRelation xmi:id="id.6" xmi:type="action:Calls" to="id.23" from="id.5"/>
          </codeElement>
          <actionRelation xmi:id="id.7" xmi:type="action:Flow" to="id.5" from="id.4"/>
          <actionRelation xmi:id="id.8" xmi:type="action:ExceptionFlow"
                          to="id.10" from="id.4"/>
          <actionRelation xmi:id="id.9" xmi:type="action:ExitFlow" to="id.17" from="id.4"/>
        </codeElement>
        <codeElement xmi:id="id.10" xmi:type="action:CatchUnit" name="c1">
          <codeElement xmi:id="id.11" xmi:type="code:ParameterUnit" name="e" type="id.67"/>
          <codeElement xmi:id="id.12" xmi:type="action:ActionElement" name="a2" kind="Call">
            <codeElement xmi:id="id.13" xmi:type="code:Value"
                         name="&amp;quot;Something went wrong&amp;quot;" type="id.69"/>
            <actionRelation xmi:id="id.14" xmi:type="action:Reads" to="id.13" from="id.12"/>
            <actionRelation xmi:id="id.15" xmi:type="action:Calls" to="id.66" from="id.12"/>
          </codeElement>
          <actionRelation xmi:id="id.16" xmi:type="action:Flow" to="id.12" from="id.10"/>
        </codeElement>
        <codeElement xmi:id="id.17" xmi:type="action:FinallyUnit" name="f1">
          <codeElement xmi:id="id.18" xmi:type="action:ActionElement" name="a3" kind="Call">
            <codeElement xmi:id="id.19" xmi:type="code:Value"
                         name="&amp;quot;Good bye&amp;quot;" type="id.69"/>
            <actionRelation xmi:id="id.20" xmi:type="action:Reads" to="id.19" from="id.18"/>
            <actionRelation xmi:id="id.21" xmi:type="action:Calls" to="id.66" from="id.18"/>
          </codeElement>
          <actionRelation xmi:id="id.22" xmi:type="action:Flow" to="id.18" from="id.17"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.23" xmi:type="code:MethodUnit" name="bar">
        <entryFlow xmi:id="id.24" to="id.25" from="id.23"/>
        <codeElement xmi:id="id.25" xmi:type="action:TryUnit" name="t2">
          <codeElement xmi:id="id.26" xmi:type="action:ActionElement"
                       name="a4" kind="ArrayReplace">
            <source xmi:id="id.27" language="Java" snippet="arr[20]=20"/>
            <codeElement xmi:id="id.28" xmi:type="code:Value" name="20" type="id.70"/>
            <actionRelation xmi:id="id.29" xmi:type="action:Addresses"
                        to="id.59" from="id.26"/>
            <actionRelation xmi:id="id.30" xmi:type="action:Reads" to="id.28" from="id.26"/>
            <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.28" from="id.26"/>
            <actionRelation xmi:id="id.32" xmi:type="action:Writes" to="id.61" from="id.26"/>
            <actionRelation xmi:id="id.33" xmi:type="action:Flow" to="id.34" from="id.26"/>
          </codeElement>
          <codeElement xmi:id="id.34" xmi:type="action:ActionElement" name="a5" kind="Call">
            <actionRelation xmi:id="id.35" xmi:type="action:Reads" to="id.59" from="id.34"/>
            <actionRelation xmi:id="id.36" xmi:type="action:Calls" to="id.66" from="id.42"/>
          </codeElement>
          <actionRelation xmi:id="id.37" xmi:type="action:Flow" to="id.26" from="id.25"/>
          <actionRelation xmi:id="id.38" xmi:type="action:ExceptionFlow"
```

```
                        to="id.40" from="id.25"/>
          <actionRelation xmi:id="id.39" xmi:type="action:ExitFlow"/>
        </codeElement>
        <codeElement xmi:id="id.40" xmi:type="action:CatchUnit" name="c2">
          <codeElement xmi:id="id.41" xmi:type="code:ParameterUnit" name="e" type="id.68"/>
          <codeElement xmi:id="id.42" xmi:type="action:ActionElement" name="a6" kind="Call">
            <codeElement xmi:id="id.43" xmi:type="code:Value"
                         name="&amp;quot;Oops&amp;quot;" type="id.69"/>
            <actionRelation xmi:id="id.44" xmi:type="action:Reads" to="id.43" from="id.47"/>
            <actionRelation xmi:id="id.45" xmi:type="action:Calls" to="id.66" from="id.42"/>
            <actionRelation xmi:id="id.46" xmi:type="action:Flow" to="id.47" from="id.42"/>
          </codeElement>
          <codeElement xmi:id="id.47" xmi:type="action:ActionElement" name="a7" kind="Throw">
            <codeElement xmi:id="id.48" xmi:type="code:Value"
                         name="&amp;quot;Went too far&amp;quot;" type="id.69"/>
            <actionRelation xmi:id="id.49" xmi:type="action:Reads" to="id.48" from="id.47"/>
            <actionRelation xmi:id="id.50" xmi:type="action:Throws"/>
          </codeElement>
          <actionRelation xmi:id="id.51" xmi:type="action:Flow" to="id.42" from="id.40"/>
        </codeElement>
        <codeElement xmi:id="id.52" xmi:type="action:FinallyUnit" name="f2">
          <codeElement xmi:id="id.53" xmi:type="action:ActionElement" name="a8" kind="Call">
            <actionRelation xmi:id="id.54" xmi:type="action:Reads" to="id.59" from="id.53"/>
            <actionRelation xmi:id="id.55" xmi:type="action:Calls" to="id.66" from="id.42"/>
          </codeElement>
          <actionRelation xmi:id="id.56" xmi:type="action:Flow" to="id.53" from="id.52"/>
        </codeElement>
        <codeElement xmi:id="id.57" xmi:type="code:Signature">
          <parameterUnit xmi:id="id.58" type="id.63" kind="throws"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.59" xmi:type="code:MemberUnit"
                   name="arr" type="id.60" size="10">
        <codeElement xmi:id="id.60" xmi:type="code:ArrayType">
          <itemUnit xmi:id="id.61" type="id.70"/>
          <indexUnit xmi:id="id.62" type="id.70"/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.63" xmi:type="code:ClassUnit"
                 name="MoreDescriptiveException" isAbstract="true">
      <codeRelation xmi:id="id.64" xmi:type="code:Extends" to="id.67" from="id.63"/>
    </codeElement>
  </model>
  <model xmi:id="id.65" xmi:type="code:CodeModel" name="Java common definitions">
    <codeElement xmi:id="id.66" xmi:type="code:CallableUnit" name="println"/>
    <codeElement xmi:id="id.67" xmi:type="code:ClassUnit" name="Exception"/>
    <codeElement xmi:id="id.68" xmi:type="code:ClassUnit"
                 name="ArrayIndexOutOfBoundsException" isAbstract="false"/>
    <codeElement xmi:id="id.69" xmi:type="code:StringType"/>
    <codeElement xmi:id="id.70" xmi:type="code:IntegerType"/>
  </model>
</kdm:Segment>
```

13.9 ExceptionFlow Class Diagram

ExceptionFlow class diagram defines meta-model elements that represent flow of control determined by exception handling mechanisms common to several programming languages.

Because of their somewhat unpredictable nature, exceptions can create havoc in performing flow analysis. The concepts here support tracking both user-defined exceptions, normal system exceptions, and runtime exceptions.

There exists certain action elements (for example, representing statements) where the exceptions are raised and that throw the exception and initiate the transfer of control to the exception flow. The flow for these is represented by the ExceptionFlow class. The ExceptionFlow relationship goes from an ActionElement representing the source of the throw, to a CallableElement that represents the catcher of the exception. The ExceptionFlow target is either the local CatchUnit that will handle the exception or point back to the TryUnit.

Exception flow elements are **optional** for L0 KDM models. KDM export tools at L0 compliance level may lack the full understanding necessary to precisely pinpoint the source of an Exception. KDM model produced by one tool can be further analyzed by a different tool to add more information about the flow of control determined by the exception handling mechanism. The origin of the ExceptionFlow can be under specified and use the TryUnit as the “from” point for the ExceptionFlow, thus omitting the details of exactly where the exception might have been raised. For user-defined exceptions, tools should normally be able to capture the origin points by analyzing the various methods invoked and recursively analyzing their code and also by taking advantage of the declaration throws clause (except that this might list exceptions that are not always thrown). As for system/runtime exceptions, it is really up to the implementer to determine how much they want to capture. Knowing what type of operations can cause those kinds of exceptions can go a long way in supporting complex analysis.

For each exception that a method invocation can raise, there should be an ExceptionFlow from that point to the immediate catcher, unless the user chose to only represent the exception generically from the TryUnit. In those cases there should be as many ExceptionFlow as there are possible exceptions that can be raised by the code in the try block.

Next if there is a finally clause, a finally flow would go from the TryUnit to the FinallyUnit to cover the finalization. The FinallyFlow is represented with a general ExitFlow relationship. This concept might appear to be a bit convoluted since normal flow would go from the end of the try and flow to the finally block and from there to the next block, but the process when we are analyzing the exception flow is as follows: For each TryUnit that the ExceptionFlow must unwind to reach its destination, it must check for an ExitFlow in the TryUnit and run through that flow before unwinding the call stack. This is repeated until the CatchUnit is reached (either it was local and already the endpoint of the ExceptionFlow or it is determined by analysis during unwinding). Then the flow for the catch is performed, followed by an ExitFlow local to the Catch, if one exists. This is as consistent as possible with the separate mechanism used by exception supporting languages to handle try/catch functionality in the first place. As stated by Bjarne Stroustrup, “The exception handling mechanism is a non local control structure based on stack unwinding that can be seen as an alternative return mechanism.” Hence the necessity for the extra level of smarts needed to analyze those flow paths.

Exceptions raised and caught within exceptions catch blocks should create and manage their own flow path.

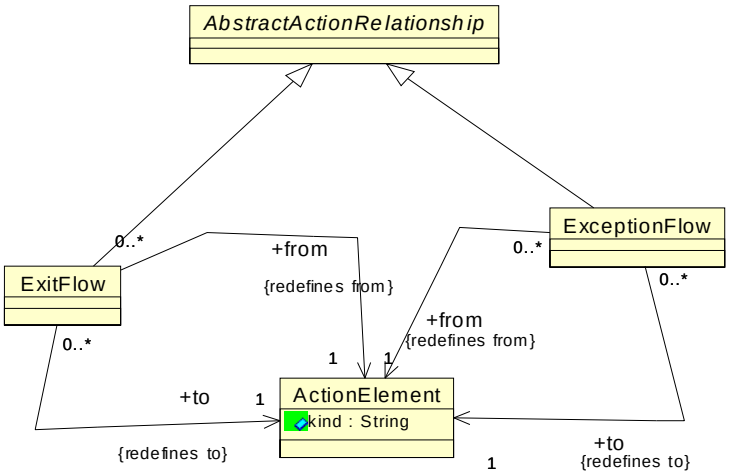

Figure 13.7 - ExceptionFlow Class Diagram

13.9.1 ExitFlow Class

ExitFlow class is a meta-model element that represents an implicit flow of control that should be taken when the corresponding block is exiting and has terminated normally or abruptly after executing the catch-block. For example, ExitFlow can be used to relate a try-block with the corresponding finally-block.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	ActionElement (for example, a try-block) for which the “on-exit” behavior was specified.
to:ActionElement[1]	ActionElement (usually, a finally-block) that represents the behavior that is invoked upon successful exit of the origin block (“on exit”).

Semantics

ExitFlow relationship represents an association between a TryUnit and the corresponding FinallyBlock according to the semantics of the programming language of the existing software system.

Example

See example in section ExceptionBlocks.

13.9.2 ExceptionFlow Class

The ExceptionFlow relationship represents an exception flow relationship between a TryUnit and the corresponding CatchUnit, or between a particular action element that can raise an exception to the corresponding CatchUnit.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	the origin of the exception flow
to:ActionElement[1]	The CatchUnit to which control is transferred when an exception is raised.

Constraints

1. The target action element of the ExceptionFlow relationship should be a CatchUnit.

Semantics

ExceptionFlow relationship represents an exception control flow between an action element that raises a certain exception, and the CatchUnit that handles this exception according to the semantics of the programming language of the existing software system.

13.10 ExceptionRelations Class Diagram

The ExceptionRelations class diagram defines a set of meta-model elements that represent data flow relationships associated with exception handling mechanism common to various programming languages.

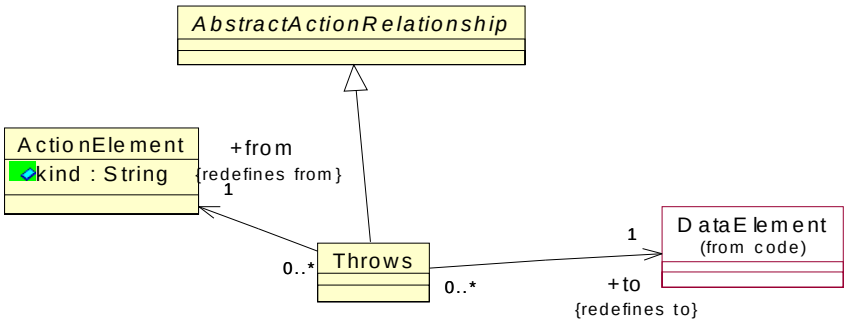

Figure 13.8 - ExceptionRelations Class Diagram

13.10.1 Throws Class

The Throws class is a meta-model element that represents throw-statements supported by several programming languages. These are the user-defined throws for exception, as opposed to the so-called normal system exceptions and runtime exceptions. Throw statements are essentially regular code elements, which simply have an additional relationship to their throw entity, using the Throws relationship.

See ExceptionBlocks and ExceptionFlow for representation of the control flow, related to exception handling mechanism.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The ActionElement that throws the exception.
to:DataElement[1]	The exception data element being thrown.

Semantics

Throws relationship represents an association between an action element that raises a certain exception and the data element that is associated with that exception. The implementer shall identify and represent these associations according to the semantics of the programming language of the existing software system.

13.11 InterfaceRelations Class Diagram

The InterfaceRelations class diagram defines KDM relationships that represent the usages of a “declarations” by the action elements. The classes and associations of the InterfaceRelations diagram are shown in Figure 13.9.

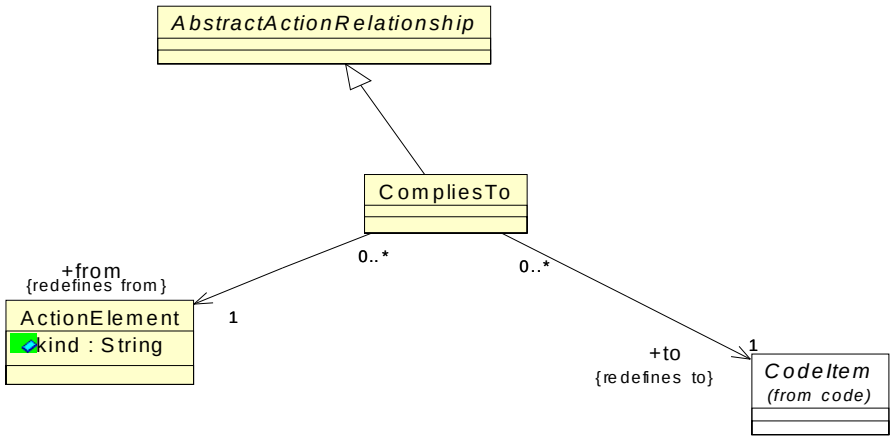

Figure 13.9 - InterfaceRelations Class Diagram

13.11.1 CompliesTo Class

The CompliesTo is a meta-model element that represents an association between an action element that “uses” some computational object, and the “declaration” of that computational object.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The origin of the relationship; action element that “uses” some computational object.
to:CodeItem[1]	the “declaration” of that computational object

Constraints

1. The kind attribute of the CodeItem at the target of the CompliesTo relationship should be equal to “external” or “abstract.”
2. The action element that is the origin of the “CompliesTo” relationship should own a callable or data action relationship to some computational object and the target of CompliesTo relationship should be one of the declarations of that computational object.

Semantics

See the InterfaceRelations section of the Code package chapter.

13.12 UsesRelations Class Diagram

The UsesRelations class diagram defines meta-model elements that represent associations between ActionElement and Datatype related to type cast operations. The class diagram shown in Figure 13.10 captures these classes and their relations.

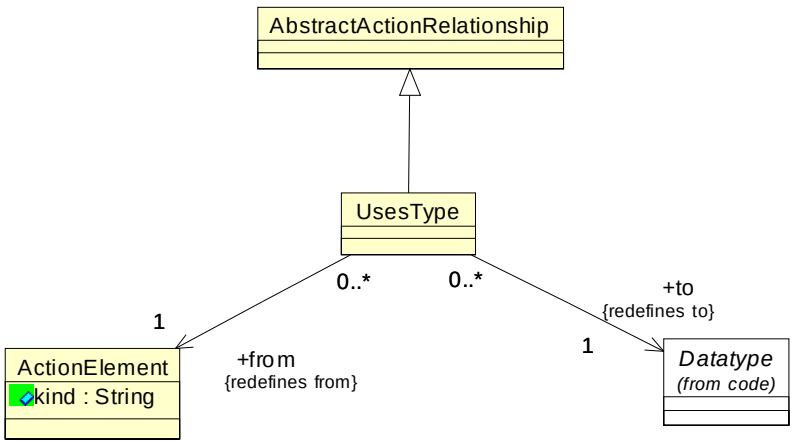

Figure 13.10 - UsesRelations Class Diagram

13.12.1 UsesType Class

The UsesType relationship represents a type cast or a type conversion performed by an ActionElement.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	The action element that performs a type cast or a type conversion.
to:Datatype[1]	The datatype involved in a type operation.

Semantics

UsesType relationship represents an association between an action element that performs a type cast or a type conversion operation and the corresponding Datatype. The precise nature of the type operation can be further specified by the “kind” attribute of the action element. See the “micro KDM” chapter.

13.13 ExtendedActionElements Class Diagram

The ExtendedActionElements class diagram defines an additional “wildcard” generic element for the code model as determined by the KDM model pattern: a generic action relationship.

The classes and associations of the ExtendedActionElements diagram are shown in Figure 13.11.

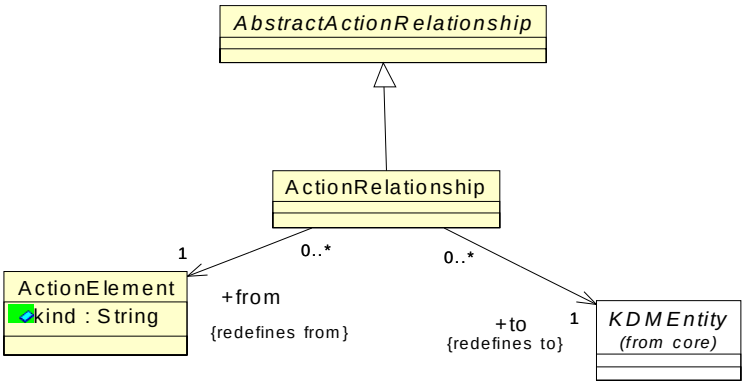

Figure 13.11 - ExtendedActionElements Class Diagram

13.13.1 ActionRelationship Class (generic)

The ActionRelationship is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractActionRelationship

Associations

from:ActionElement[1]	the origin action element
to:KDMEntity[1]	the target KDM entity

Constraints

1. ActionRelationship should have at least one stereotype.

Semantics

An action relationship with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model relationship types of the code model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

14 Micro KDM

This chapter describes the foundation for KDM models with precise semantics, referred to as the L1 MICRO KDM compliance point (“micro KDM”).

Let’s use the term KDM “macro Action” to refer to an ActionElement with under specified semantics, whose role in a KDM meta-model is to represent certain existing behavior corresponding to one or more statements in the existing software system. The term “macro action” should not be confused with the preprocessor directives common to certain programming languages.

A KDM “macro action” represents the endpoint of the action relationships, and can own a SourceRef element, which links it to the artifacts of the existing software system. KDM L0 compliance point does not specify the semantics of a KDM “macro action” leaving it up to implementers to design the mapping from the programming languages involved in the artifacts of the existing software systems and KDM.

In order to use the KDM Program Element layer for control- and data-flow analysis applications, as well as for providing more precision for the Resource Layer and the abstraction Layer, additional semantics constraints are required to coordinate producers and consumers of KDM models. The micro KDM compliance point serves this purpose. It constrains the granularity of the leaf action elements, and their meaning by providing the set of micro actions with predefined semantics.

According to the “micro KDM” approach, the original “macro action” is treated as a container that owns certain “micro actions” with predefined semantics and thus precise semantics of the “macro action” is defined. As the result micro KDM constrains the patterns of how to map the statements of the existing system as determined by the programming language into KDM. This is similar to the mapping performed by a compiler into the Java Virtual Machine or Microsoft .NET.

From a compiler technology perspective, KDM micro actions provide the so-called Intermediate Representation (IR) for control and data flow analysis. Micro KDM is a rather high-level IR. Micro KDM actions are aligned with the ISO 11404 datatypes (operations on primitive datatypes and access to complex datatypes) as well as with the KDM patterns for representing program elements.

Separation into a “macro action” and “micro actions” allows:

- The flexibility of selecting the “macro action element” as the point for linking it to the existing artifacts. For example to a source file or to an AST, providing a meaningful source ref (a macro action can still represent one or more statements in the original existing system), and
- provides precise representation of the original “macro action” through a mapping to micro actions with predefined semantics.

Micro actions are the actual endpoints of the KDM relationships. The original “macro action element” (the one that owns the micro actions) aggregates these relationships through the uniform semantics of KDM aggregated relationships. The micro actions fit into the existing flow semantics, which makes it possible to use KDM representations for precise control and data flow analysis.

According to the “encapsulated relationship pattern” of KDM, each action element should own the ActionRelationships that originate from this action element. A KDM relationship originates from an element, if the “from” property of the relationship is equal to the identity of the element. The encapsulated relationship pattern is described in the Code KDM section. The collection of action relationships owned by an element is ordered. From the KDM perspective, the owned ActionRelationships represent the parameters to the action element.

Each micro KDM action has the following 5 parts.

1. Action Kind - is nature of the operation performed by the micro action. This is represented as a “kind” attribute to the micro action. The action kind may designate certain outgoing relationships as part of the Control. For example, the “Call” micro action designates the Calls outgoing relationship as part of Control. Action kinds are defined as case sensitive strings in Annex A.
2. Outputs - represented by the owned outgoing Writes relationship, which usually represents the result of the micro action. This part is optional.
3. Inputs - Ordered outgoing Reads and/or Addresses relationships that are owned by the action element, the order of the relationships represent the order of the arguments for a micro action.
4. Control part - owned outgoing control flow relationships for the action.
5. Extras part - owned relationships other than Reads, Writes, and not designated as part of Control by the action Kind. For example, these can be interface compliance relation “CompliesTo” or any extended action relationships.

Constraints

1. Every leaf action element of the KDM model shall be a micro KDM action, where the operation performed by the action is designated by the value of the action kind, specified in the list of the micro actions in Annex A.
2. Implementers shall capture the meaning of the existing artifacts as determined by the programming languages and runtime platform and map it into connected graphs of micro actions; the meaning of the resulting micro KDM model is determined by the semantics of the micro actions.

Semantics

Semantics of KDM micro actions is defined in Annex A: “Semantics of the micro KDM action elements.”

Example

```
z=1+f(x,y);
*d[x+3]=1;
d[y+3]=&z;
y=*d[x+3];
```

function foo does not comply to micro KDM semantic constraints;
function bar complies to micro KDM

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment
        xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="Micro KDM Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
     <codeElement xmi:id="id.1" xmi:type="code:CodeAssembly">
       <codeElement xmi:id="id.2" xmi:type="code:CallableUnit" name="foo" kind="regular">
          <entryFlow xmi:id="id.3" to="id.4" from="id.2"/>
          <codeElement xmi:id="id.4" xmi:type="action:ActionElement" name="f1" kind="unknown">
             <source xmi:id="id.5" language="C" snippet="z=1+f(x,y)"/>
             <actionRelation xmi:id="id.6" xmi:type="action:Calls" to="id.107" from="id.4"/>
             <actionRelation xmi:id="id.7" xmi:type="action:Reads" to="id.97" from="id.4"/>
```

```
      <actionRelation xmi:id="id.8" xmi:type="action:Reads" to="id.98" from="id.4"/>
      <actionRelation xmi:id="id.9" xmi:type="action:Writes" to="id.99" from="id.4"/>
      <actionRelation xmi:id="id.10" xmi:type="action:Reads" to="id.105" from="id.4"/>
      <actionRelation xmi:id="id.11" xmi:type="action:Flow" from="id.4"/>
    </codeElement>
    <codeElement xmi:id="id.12" xmi:type="action:ActionElement" name="f2" kind="unknown">
      <source xmi:id="id.13" language="C" snippet="*d[x+3]=1;d[y+3]=&amp;amp;z;y=*d[x+3];"/>
      <actionRelation xmi:id="id.14" xmi:type="action:Reads" to="id.97" from="id.12"/>
      <actionRelation xmi:id="id.15" xmi:type="action:Addresses" to="id.100" from="id.12"/>
      <actionRelation xmi:id="id.16" xmi:type="action:Reads" to="id.106" from="id.12"/>
      <actionRelation xmi:id="id.17" xmi:type="action:Reads" to="id.105" from="id.12"/>
      <actionRelation xmi:id="id.18" xmi:type="action:Addresses" to="id.100" from="id.12"/>
      <actionRelation xmi:id="id.19" xmi:type="action:Reads" to="id.98" from="id.12"/>
      <actionRelation xmi:id="id.20" xmi:type="action:Reads" to="id.106" from="id.12"/>
      <actionRelation xmi:id="id.21" xmi:type="action:Addresses" to="id.99" from="id.12"/>
      <actionRelation xmi:id="id.22" xmi:type="action:Writes" to="id.98" from="id.4"/>
      <actionRelation xmi:id="id.23" xmi:type="action:Addresses" to="id.100" from="id.12"/>
      <actionRelation xmi:id="id.24" xmi:type="action:Reads" to="id.97" from="id.12"/>
      <actionRelation xmi:id="id.25" xmi:type="action:Reads" to="id.106" from="id.12"/>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.26" xmi:type="code:CallableUnit" name="bar" kind="regular">
    <entryFlow xmi:id="id.27" to="id.28" from="id.26"/>
    <codeElement xmi:id="id.28" xmi:type="action:ActionElement" name="b1" kind="compound">
      <source xmi:id="id.29" language="C" snippet="z=1+f(x,y)"/>
      <codeElement xmi:id="id.30" xmi:type="code:StorableUnit" name="t1"
                type="id.112" kind="register"/>
      <codeElement xmi:id="id.31" xmi:type="action:ActionElement" name="b1.1" kind="Call">
        <actionRelation xmi:id="id.32" xmi:type="action:Calls" to="id.107" from="id.28"/>
        <actionRelation xmi:id="id.33" xmi:type="action:Reads" to="id.97" from="id.28"/>
        <actionRelation xmi:id="id.34" xmi:type="action:Reads" to="id.98" from="id.28"/>
        <actionRelation xmi:id="id.35" xmi:type="action:Writes" to="id.30" from="id.31"/>
        <actionRelation xmi:id="id.36" xmi:type="action:Flow" from="id.31"/>
      </codeElement>
      <codeElement xmi:id="id.37" xmi:type="action:ActionElement" name="b1.2" kind="Add">
        <actionRelation xmi:id="id.38" xmi:type="action:Reads" to="id.105" from="id.37"/>
        <actionRelation xmi:id="id.39" xmi:type="action:Reads" to="id.30" from="id.37"/>
        <actionRelation xmi:id="id.40" xmi:type="action:Writes" to="id.99" from="id.37"/>
      </codeElement>
      <actionRelation xmi:id="id.41" xmi:type="action:Flow" to="id.31" from="id.28"/>
    </codeElement>
    <codeElement xmi:id="id.42" xmi:type="action:ActionElement" name="b2" kind="compound">
      <source xmi:id="id.43" language="C" snippet="*d[x+3]=1;d[y+3]=&amp;amp;z;y=*d[x+3];"/>
      <codeElement xmi:id="id.44" xmi:type="code:StorableUnit" name="t2"
                type="id.103" kind="register"/>
      <codeElement xmi:id="id.45" xmi:type="code:StorableUnit" name="t3"
                type="id.112" kind="register"/>
      <codeElement xmi:id="id.46" xmi:type="code:StorableUnit" name="t4"
                type="id.112" kind="register"/>
      <codeElement xmi:id="id.47" xmi:type="code:StorableUnit" name="t5"
                type="id.103" kind="register"/>
```

```
 <codeElement xmi:id="id.48" xmi:type="code:StorableUnit" name="t6"
           type="id.112" kind="register"/>
 <codeElement xmi:id="id.49" xmi:type="code:StorableUnit" name="t7"
           type="id.103" kind="register"/>
 <codeElement xmi:id="id.50" xmi:type="action:ActionElement" name="b2.1" kind="Add">
    <actionRelation xmi:id="id.51" xmi:type="action:Reads" to="id.97" from="id.50"/>
    <actionRelation xmi:id="id.52" xmi:type="action:Reads" to="id.106" from="id.50"/>
    <actionRelation xmi:id="id.53" xmi:type="action:Writes" to="id.44" from="id.50"/>
    <actionRelation xmi:id="id.54" xmi:type="action:Flow" to="id.55" from="id.50"/>
 </codeElement>
<codeElement xmi:id="id.55" xmi:type="action:ActionElement" name="b2.2" kind="ArraySelect">
    <actionRelation xmi:id="id.56" xmi:type="action:Addresses" to="id.100" from="id.55"/>
    <actionRelation xmi:id="id.57" xmi:type="action:Reads" to="id.102" from="id.55"/>
    <actionRelation xmi:id="id.58" xmi:type="action:Reads" to="id.44" from="id.55"/>
    <actionRelation xmi:id="id.59" xmi:type="action:Writes" to="id.45" from="id.55"/>
    <actionRelation xmi:id="id.60" xmi:type="action:Flow" from="id.55"/>
 </codeElement>
 <codeElement xmi:id="id.61" xmi:type="action:ActionElement" name="b2.3" kind="PtrReplace">
    <actionRelation xmi:id="id.62" xmi:type="action:Reads" to="id.45" from="id.61"/>
    <actionRelation xmi:id="id.63" xmi:type="action:Reads" to="id.105" from="id.61"/>
    <actionRelation xmi:id="id.64" xmi:type="action:Writes" to="id.104" from="id.61"/>
    <actionRelation xmi:id="id.65" xmi:type="action:Flow" to="id.66" from="id.61"/>
 </codeElement>
 <codeElement xmi:id="id.66" xmi:type="action:ActionElement" name="b2.4" kind="Add">
    <actionRelation xmi:id="id.67" xmi:type="action:Reads" to="id.98" from="id.12"/>
    <actionRelation xmi:id="id.68" xmi:type="action:Reads" to="id.106" from="id.12"/>
    <actionRelation xmi:id="id.69" xmi:type="action:Writes" to="id.46" from="id.66"/>
    <actionRelation xmi:id="id.70" xmi:type="action:Flow" to="id.71" from="id.66"/>
 </codeElement>
 <codeElement xmi:id="id.71" xmi:type="action:ActionElement" name="b2.5" kind="Ptr">
    <actionRelation xmi:id="id.72" xmi:type="action:Addresses" to="id.99" from="id.12"/>
    <actionRelation xmi:id="id.73" xmi:type="action:Writes" to="id.47" from="id.71"/>
    <actionRelation xmi:id="id.74" xmi:type="action:Flow" to="id.75" from="id.71"/>
 </codeElement>
<codeElement xmi:id="id.75" xmi:type="action:ActionElement" name="b2.6" kind="ArrayReplace">
    <actionRelation xmi:id="id.76" xmi:type="action:Addresses" to="id.100" from="id.12"/>
    <actionRelation xmi:id="id.77" xmi:type="action:Reads" to="id.46" from="id.75"/>
    <actionRelation xmi:id="id.78" xmi:type="action:Reads" to="id.47" from="id.75"/>
    <actionRelation xmi:id="id.79" xmi:type="action:Writes" to="id.102" from="id.75"/>
    <actionRelation xmi:id="id.80" xmi:type="action:Flow" from="id.75"/>
 </codeElement>
 <codeElement xmi:id="id.81" xmi:type="action:ActionElement" name="b2.7" kind="Add">
    <actionRelation xmi:id="id.82" xmi:type="action:Reads" to="id.97" from="id.12"/>
    <actionRelation xmi:id="id.83" xmi:type="action:Reads" to="id.106" from="id.12"/>
    <actionRelation xmi:id="id.84" xmi:type="action:Writes" to="id.48" from="id.81"/>
    <actionRelation xmi:id="id.85" xmi:type="action:Flow" from="id.81"/>
 </codeElement>
 <codeElement xmi:id="id.86" xmi:type="action:ActionElement" name="b2.8" kind="ArraySelect">
    <actionRelation xmi:id="id.87" xmi:type="action:Addresses" to="id.100" from="id.12"/>
    <actionRelation xmi:id="id.88" xmi:type="action:Reads" to="id.48" from="id.86"/>
    <actionRelation xmi:id="id.89" xmi:type="action:Reads" to="id.102" from="id.86"/>
```

```
            <actionRelation xmi:id="id.90" xmi:type="action:Writes" to="id.49" from="id.86"/>
            <actionRelation xmi:id="id.91" xmi:type="action:Flow" to="id.92" from="id.86"/>
          </codeElement>
          <codeElement xmi:id="id.92" xmi:type="action:ActionElement" name="b2.9" kind="PtrSelect">
            <actionRelation xmi:id="id.93" xmi:type="action:Reads" to="id.49" from="id.92"/>
            <actionRelation xmi:id="id.94" xmi:type="action:Reads" to="id.104" from="id.92"/>
            <actionRelation xmi:id="id.95" xmi:type="action:Writes" to="id.98" from="id.92"/>
          </codeElement>
          <actionRelation xmi:id="id.96" xmi:type="action:Flow" to="id.50" from="id.42"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.97" xmi:type="code:StorableUnit" name="x" type="id.112"/>
      <codeElement xmi:id="id.98" xmi:type="code:StorableUnit" name="y" type="id.112"/>
      <codeElement xmi:id="id.99" xmi:type="code:StorableUnit" name="z" type="id.112"/>
      <codeElement xmi:id="id.100" xmi:type="code:StorableUnit" name="d" type="id.101">
        <codeElement xmi:id="id.101" xmi:type="code:ArrayType" name="">
          <itemUnit xmi:id="id.102" name="d[]" type="id.103">
            <codeElement xmi:id="id.103" xmi:type="code:PointerType">
              <itemUnit xmi:id="id.104" name="*d[]" type="id.112"/>
            </codeElement>
          </itemUnit>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.105" xmi:type="code:Value" name="1" type="id.112"/>
      <codeElement xmi:id="id.106" xmi:type="code:Value" name="3" type="id.112"/>
      <codeElement xmi:id="id.107" xmi:type="code:CallableUnit" name="f" type="id.108">
        <codeElement xmi:id="id.108" xmi:type="code:Signature">
          <parameterUnit xmi:id="id.109" name="a" type="id.112" pos="1"/>
          <parameterUnit xmi:id="id.110" name="b" type="id.112" pos="2"/>
          <parameterUnit xmi:id="id.111" type="id.112" kind="return"/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.112" xmi:type="code:IntegerType" name="int"/>
  </model>
</kdm:Segment>
```

Part III - Runtime Resources Layer

This section describes common patterns for representing the operating environment of existing software systems. The following are the common properties of the Resource Layer packages Data, UI, Platform, and Event:

- They provide modeling elements to represent “resources” (something managed by the runtime platform).
- They provide abstract “resource actions” to manage these resources.
- These actions are implemented by the program elements as one or more API calls to some external platform-specific packages.
- There is a binding involved between the actions and the resources.
- Resource may involve some “inverted” control in the form of callbacks and event handlers, allowing applications to be programmed in event-driven style.
- The content of the information flow involving the resource is associated with some data organization.
- Resource often has a certain state, and tracking the changes of the state over time may be an important concern in understanding the logic of the existing system.

Since Resource Layer packages capture high-value knowledge about the existing system and its operating environment, which may involve advance analysis and some manual expertise KDM is designed in such a way that the Resource level analysis can use KDM models from the Platform Elements Layer as input and produce Resource Layer models as output. There should be no references from lower KDM layers to higher layers; therefore, new Resource Layer models can be built on top of existing Program Element layer models.

Resource layer package systematically uses the following KDM patterns:

- Each Resource Layer package defines entities and containers to represent specific “resources.” Each package may define additional elements to represent additional concerns. For example, the Data package involves less resource definitions, and focuses on the representation of various data organization capabilities. The Event package provides the meta-model elements for representing state, state transitions caused by events. States, transitions, and events can be considered as runtime platform resources. The UI package provides the meta-model elements for representing user interfaces. User interfaces can also be considered runtime platform resources. The Platform package deals with conventional runtime platform resources, such as inter-process communication, the use of registries, management of data, etc.

- Each Resource Layer package defines specific structural relations between “resources.” For example, the Platform package defines relationship BindsTo, which represents a logical association between two resources.

- Each Resource Layer package defines specific resource actions to represent manipulation of resource through API calls. Resource actions use the following KDM pattern. Each resource action is an entity of the base abstract class for the corresponding package. This class is named AbstractXXXElement, where “XXX” is the name of the package. So, the resource action is not a subclass of ActionElement, and this promotes modularity between Resource Layer packages, each of which defines an independent KDM compliance point. However, each resource action owns one or more ActionElements through property called “abstraction.” Each resource action also has the property called “implementation” that uses the KDM grouping mechanism to associate the resource action with one or more ActionElements owned by the Code model. The “implementation” group of the resource action represents the original API calls as they were represented in the Program Elements layer input model. The “abstraction” property uses KDM container mechanism to add the behavior counterparts to the API calls that represent the true logic of the resource operation, including the flow of data and control. The “abstracted” ActionElements are owned directly by the corresponding resource action, and are not part of any Code model.

- The nature of the resource-specific operation performed by a particular resource action is represented by the “kind” attribute of the resource actions. The resource action owns resource action relations through the “abstraction” action container. It is the owned “abstracted” action that is the direct owner of the resource action relationship.
- “abstraction” action container property is in fact systematically added to all elements of Resource Layer packages. This way each resource can use the meta-model elements defined in the Program Elements layer to specify behavior specific to that resource.
- The “abstraction” action container pattern is used to systematically represent the forms of “inverted” control provided by runtime platforms. This pattern can be separately referred to as the KDM Event pattern. Each Resource Layer package defines its own meta-model element for representing events. For example, the UI package defines the class UIEvent. The “abstraction” action container mechanism allows ActionElements to be added to event elements. Calls relation originating from such an abstracted action element represents the “callback” mechanism, provided by several runtime platforms.

Resource Layer packages are independent, however they can offer additional capabilities when more than one is implemented. In order not to enforce any particular order in which these packages should be implemented, KDM involves the following approach: resource action relationships are subclasses of the AbstractActionRelationship class from the Action Package. In the full KDM implementation that uses all packages, all resource actions are available for any ActionElement. Code models should not use the extended relationships. The extended resource action relationships can only be used by the actions in “abstraction” action containers in Resource Layer models. A notable example of this mechanism are the actions “HasState” defined in the Event package, which makes it possible to associate an element of an event model with any resource. Another example is the “HasContent” relation defined in the Data package, which allows associating an element of a data model with any resource.

- The “abstraction” action containers, available for each resource also allow arbitrary Flow relations between “resource actions” and between resources to provide abstractions of the flow between “resource actions.”
- The Resource Layer patterns are aligned with the micro KDM, which allow precise modeling of behavior related to resources as the foundation for holistic high-fidelity analysis of existing software systems. It can be achieved by associating sets of precise micro KDM actions with “abstraction” action containers.

Additional Runtime concerns involve dynamic structures (instances of some logical entities and their relationships) that emerge at the so-called “run time” of the software system. For example, dynamic entities include processes and threads. Instances of processes and threads can be created dynamically and in many cases relations between the dynamically created instances of processes and threads are an essential part of the knowledge of existing systems. Pure logical perspective in that case may be insufficient.

When separation of concerns between application code and Runtime platform is considered, it is important to be clear about the so-called *bindings* and various mechanisms to achieve a binding or delay it.

A binding is a common way of referring to a certain irrevocable implementation decision. Too much binding is often referred to as “hardcoding.” This often results in systems that are difficult to maintain and reuse. They are often also difficult to understand. Too little binding leads to dynamic systems, where everything is resolved at run time (as late as possible). This often results in systems that are difficult to understand and error-prone. Modern platforms excel in managing binding time. Usually binding is managed at deployment time.

A large number of software development practices is about efficient management of binding time, including portable adaptors, code generation, model-driven architecture. Efficient management of binding time is often called platform independence.

Binding time

- Generation time binding
- Language & platform design binding
- Versioning time
- Compile time binding, including
  - macro expansion
  - Templates
  - Product line variants defined by conditional compilation
- Link time binding
- Deployment time binding
- Initialization time binding
- Run time

Binding Time	**What is being bound**	**Result**
Generation time	Syntax, variant, pattern, mapping, etc.	Generated code
Language & platform design	Syntax, entities and relations, including platform resource types	Source code
Versioning	Module source files	Module version
Compile time	Intra-module relations (def-use)	Module
-- Macro	Syntax, macro to expanded code	Expanded macro (source code)
-- Template	Template parameters	Template instance
-- Product line variant defined by conditional compilation and includes	Conditional compilation, macro, includes, symbolic links.	Component Variant
(static) Link time	Intra-component relations within deployable component	Deployed Component
Deployment time	Resource names to resources (using platform-specific configuration files)	Deployed System
Initialization time	Component implementation to component interface; major processes and threads; dynamic linking, dynamic load (using platform-specific configuration files).	System
Run time	User input, object factory, virtual function, function pointer, reflection, instances of processes, instances of objects, instances of data, etc.	Particular Execution Path

15 Platform Package

15.1 Overview

Platform package defines a set of meta-model elements whose purpose is to represent the runtime operating environments of existing software systems. Application code is not self-contained as it is determined not only by the selected programming languages, but also by the selected Runtime platform. Platform elements determine the execution context for the application. Platform package defines meta-model elements that represent common Runtime platform concerns:

- Runtime platform consists of many diverse elements (platform parts).
- Platform provides resources to deployment components.
- Platform provides services that are related to resources.
- Application code invokes services to manage the life-cycle of a resource.
- Control flow between application components is often determined by the platform.
- Platform provides error handling across application components.
- Platform provides integration of application components.

Examples of Platform Parts include UNIX OS File System, UNIX OS process management system, Windows 2000, OS/390, Java (J2SE), Perl language Runtime support, IBM CICS TS, IBM MQSeries, Jakarta Struts, BEA Tuxedo, CORBA, HTTP, TCP/IP, Eclipse, EJB, JMS, Database middleware, Servlets.

Platform package defines an *architectural viewpoint* for the Platform domain.

- **Concerns**:
  - What are the resources used by the software system?
  - What elements of the run-time platform are used by the software system?
  - What behavior is associated with the resources of the run-time platform?
  - What control flows are initiated by the events in the resources?
  - What control flows are initiated by the run-time environment?
  - What are the bindings to the run-time environment?
  - What are the deployment configurations of the software system?
  - What are the dynamic/concurrent threads of activity within the software system?
- **Viewpoint language**:

  Platform views conform to KDM XMI schema. The *viewpoint language* for the Platform *architectural viewpoint* is defined by the Platform package. It includes an abstract entity AbstractPlatformElement, several generic entities, such as ResourceType, RuntimeResource, as well as several concrete entities, such as PlatformAction, PlatformEvent, ExternalActor, MarshalledResource, NamingResource, etc. The viewpoint language for the Platform architectural viewpoint also includes several relationships, which are subclasses of AbstractPlatformRelationship.

- **Analytic methods**:

  The Platform architectural viewpoint supports the following main kinds of checking:

  - Data flow (for example, what action elements read from a given resource; what action elements write to a given resource; what action elements manage a given resource; including indirect data flow using a MarshalledResource or a MessagingResource where a particular resource is used to perform a data flow between the "send" action element and the "receive" action element).
  - Control flow (for example, what action elements are triggered by events in a given resource; what action elements operate on a given resource).
  - Identify of resource instances based on resource handles in various modules.

  Platform Views are used in combination with Code views and Inventory views.

- **Construction methods**:

  - Platform views that correspond to the KDM Platform architectural viewpoint are usually constructed by analyzing Code views for the given system as well as the platform-specific configuration artifacts. The platform extractor tool uses the knowledge of the API and semantics for the given run-time platform to produce one or mode Platform views as output
  - As an alternative, for some languages like Cobol, in which the elements of the run-time platform are explicitly defined by the language, the platform views are produced by the parser-like tools which take artifacts of the system as the input and produce one or mode Platform views as output (together with the corresponding Code views)
  - Construction of the Platform view is determined by the semantics of the run-time platform, and it based on the mapping from the given run-time platform to KDM; such mapping is specific only to the run-time platform and not to a specific software system
  - The mapping from a particular run-time platform to KDM may produce additional information (system-specific, or platform-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations

15.2 Organization of the Platform Package

The Platform package consists of the following 10 class diagrams.

1. PlatformModel
2. PlatformInheritances
3. PlatformResources
4. PlatformRelations
5. PlatformActions
6. ProvisioningRelations
7. Deployment
8. RuntimeResources
9. RuntimeActions
10. ExtendedPlatformElements

The Platform package depends on the following packages:

- Core
- kdm
- Code
- Action

15.3 PlatformModel Class Diagram

The PlatformModel class diagram follows the uniform pattern for extending KDM Framework followed by each KDM model. The classes and associations of the PlatformModel diagram are shown in Figure 15.1.

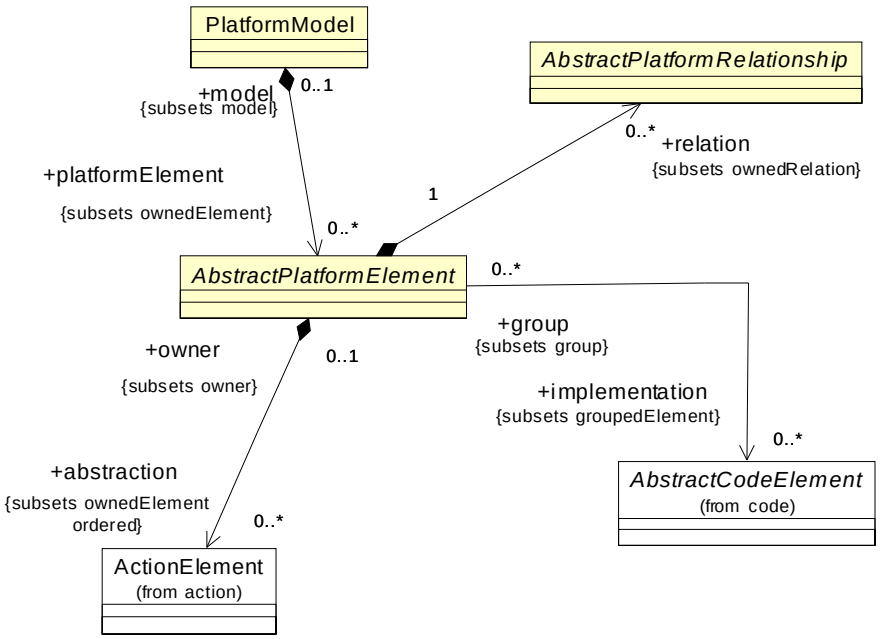

Figure 15.1 - PlatformModel Class Diagram

15.3.1 PlatformModel Class

PlatformModel is a specific KDM model that owns collections of facts about the existing software system such that these facts correspond to the Platform domain. PlatformModel provides a container for platform elements.

Superclass

KDMModel

Associations

platformElement:PlatformElement[0..*]	owned platform elements

Semantics

PlatformModel is a logical container for platform elements. The implementer shall arrange platform elements into one or more platform models.

15.3.2 AbstractPlatformElement Class (abstract)

The AbstractPlatformElement is an abstract meta-model element that represents entities of the operating environments of software systems.

Superclass

KDMEntity

Associations

platformRelation:PlatformRelation[0..*]	Platform relationship that originates from the platform element.
abstraction:ActionElement[0..*]	owned “abstraction” actions
implementation:AbstractCodeElement[0..*]	Grouped association to the AbstractCodeElement element that are represented by the current PlatformElement from some CodeModel.
source:SourceRef[0..*]	traceability links owned by the given platform element

Constraints

1. Implementation AbstractCodeElement should be owned by some CodeModel.
2. Implementation AbstractCodeElement should be subclasses of ComputationalObject or ActionElement.
3. Abstraction ActionElement should be owned by the same PlatformModel.

Semantics

An instance of an AbstractPlatformElement represents either an instance of some runtime resource or some API call that manages some runtime resource. The implementation association links AbstractPlatformElement to the corresponding elements of some CodeModel. “Abstraction” action elements can be used to specify precise semantics of the PlatformElement.

15.3.3 AbstractPlatformRelationship Class (abstract)

The AbstractPlatformRelationship is an abstract meta-model element that represents associations between platform entities.

Superclass

KDMRelationship

Semantics

An instance of an AbstractPlatformRelationship represents structural association between two runtime resources.

15.4 PlatformInheritances Class Diagram

The PlatformInheritances class diagram represents inheritances of the meta-modeling elements of the Platform package. The classes and associations of the PlatformInheritances diagram are shown in Figure 15.2.

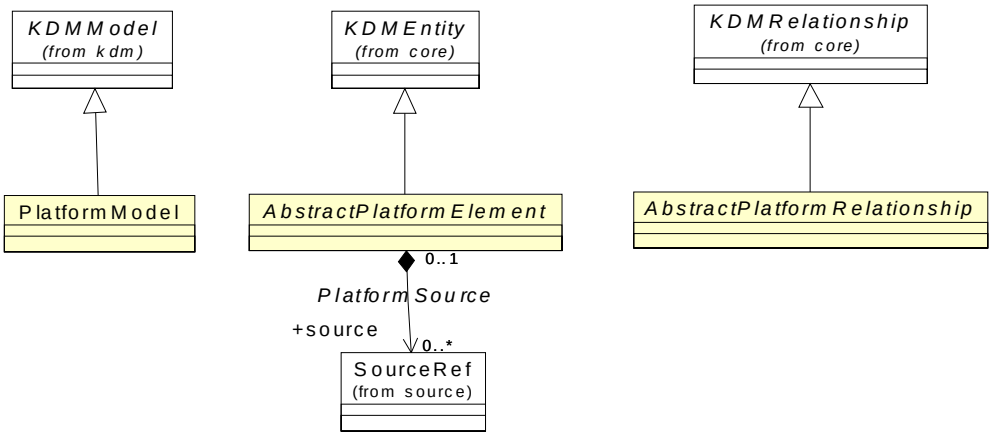

Figure 15.2 - PlatformInheritances Class Diagram

15.5 PlatformResources Class Diagram

The PlatformResources class diagram defines the meta-model elements to represent platform resources. The classes and associations of the PlatformResources diagram are shown in Figure 15.3.

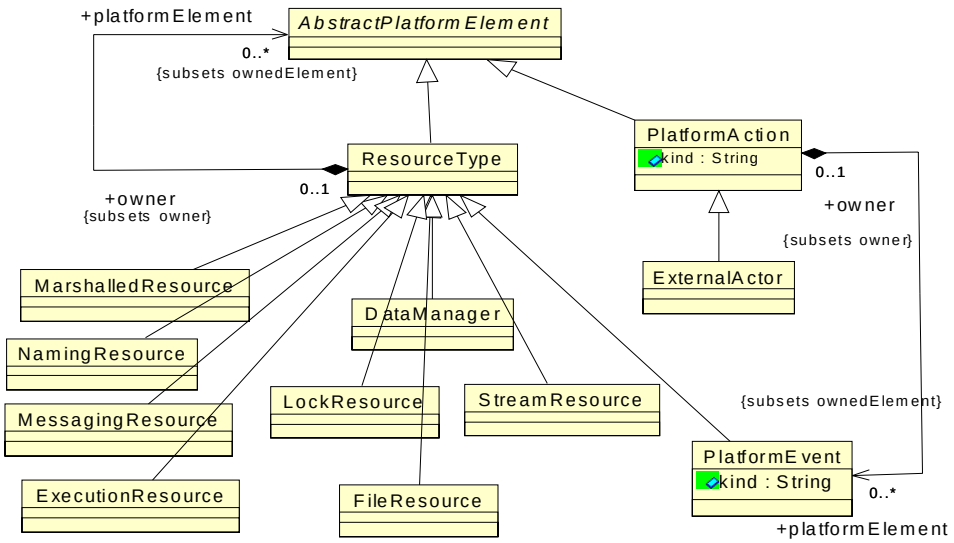

Figure 15.3 - PlatformResources Class Diagram

15.5.1 ResourceType Class

The ResourceType is a meta-model element that represents a platform resource. The purpose of a platform is to simplify application development by closing the gap between the application domain and the facilities that are available to application programmers. The latter are referred to as platform resources. Examples of resource types include UNIX File, UNIX IO Stream, UNIX socket, UNIX Process, UNIX thread, AWT widget, CICS File, CICS transaction, UNIX semaphore, UNIX shared memory segment, OS/390 VSAM file, JDBC connection, HTTP session, HTTP request, UNIX memory block, CICS commarea, COBOL file.

KDM introduces Platform Resource as an explicit abstraction in order to separate the explicit parts that are written by application programmers from parts that are provided by the platform. The underlying implementation details may be quite complex, for example, marshaled call includes client stubs, skeletons, platform-managers. The type of the Platform Resource denotes the semantics of the resource.

Platform resources can be further subdivided into smaller groups of services (resource types). Platform resources are usually grouped into platform stacks. Platform resource is an element of the overall platform used by a particular system. Complete Platform is the entire collection of platform resources used by the segments of the system. Platform resource may be associated with logical packages for a particular programming language.

Superclass

AbstractPlatformElement

Associations

platformElement:AbstractPlatformElement[0..*]	The set of platform elements that are owned by the given ResourceType.

Semantics

ResourceType may represent an individual runtime resource instance or a container for several such instances.

The implementer shall identify runtime resources used by the existing software system according to the semantics of the platform used by the existing system, resource configuration files, and other appropriate sources of information.

Specific subclasses of ResourceType define specific categories of resources available to implementers. Other types of resources can be represented by a generic instance of ResourceType meta-model element with a stereotype.

15.5.2 NamingResource Class

NamingResource represents platform resources that provide registration and lookup services (e.g., registry). In the meta-model NamingResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

15.5.3 MarshalledResource Class

MarshalledResource represents platform resources that provide intercomponent communication via remote synchronous calls (for example, RPC, CORBA method call, Java remote method invocation). In the meta-model MarshalledResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

15.5.4 MessagingResource Class

MessagingResource represents platform resources that provide intercomponent communication via asynchronous messages (e.g., IBM MQSeries messages).

Superclass

ResourceType

Semantics

15.5.5 FileResource Class

FileResource represents platform resources that provide any non-database related storage. In the meta-model the FileResource class is a subclass of ResourceType. It also implements the DataInterface so that this class can be the endpoint of Data relations.

Superclass

ResourceType

Semantics

15.5.6 ExecutionResource Class

ExecutionResource represents dynamic Runtime elements (e.g., process or thread).

Superclass

ResourceType

Semantics

15.5.7 LockResource Class

LockResource represents a synchronization resource common to multithreaded runtime environments.

Superclass

ResourceType

Semantics

15.5.8 StreamResource Class

StreamResource represents a simple input/output resource, for example UNIX-like stream.

Superclass

ResourceType

Semantics

15.5.9 DataManager Class

DataManager represents a database management system. DataManager is associated with particular data elements that represent the data description of the data managed by the data manager.

Superclass

ResourceType

Semantics

15.5.10 PlatformEvent Class

The PlatformEvent class is a meta-model element representing various events and callbacks associated with runtime platforms. This class follows the KDM event pattern, common to Resource Layer packages.

Superclass

ResourceType

Attributes

kind:String	Represents the nature of the action performed by this Event.

Semantics

15.5.11 PlatformAction Class

PlatformAction class follows the pattern of a “resource action” class, specific to the Platform package. The nature of the action represented by a particular element is designated by its “kind” attribute.

Superclass

AbstractPlatformElement

Attributes

kind:String	Represents the nature of the action performed by this element.

Associations

platformElement:PlatformEvent[0..*]	The set of platform events that are owned by the given PlatformAction.

15.5.12 ExternalActor Class

ExternalActor is a meta-model element that represents entities outside of the boundary of the software system being modeled. For example, external actor can be a user of the system. External actors interact with the software system.

In the meta-model ExternalActor is a PlatformElement. Semantics of ExternalActors is outside of the scope of KDM.

Superclass

PlatformAction

Semantics

15.6 PlatformRelations Class Diagram

The PlatformRelations class diagram defines associations between ResourceTypes. The classes and associations of the PlatformRelations diagram are shown in Figure 15.4.

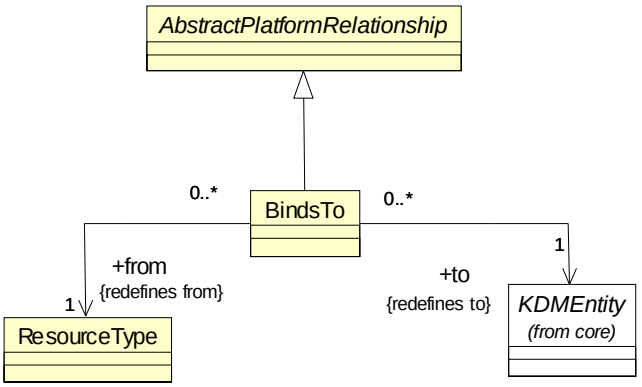

Figure 15.4 - PlatformRelations Class Diagram

15.6.1 BindsTo Class

BindsTo defines a semantic association between two ResourceTypes.

Superclass

PlatformRelationship

Associations

from:ResourceType[1]	The ResourceType that is the source of the relationship (the from-endpoint).
to:KDMEntity[1]	The KDMEntity to which the current Resource is bound (the to-endpoint).

Semantics

15.7 ProvisioningRelations Class Diagram

The ProvisioningRelations class diagram defines meta-model elements that represent the physical elements of the Runtime platform that provide certain services and manage resources.

The classes and associations of the ProvisioningRelations diagram are shown in Figure 15.5.

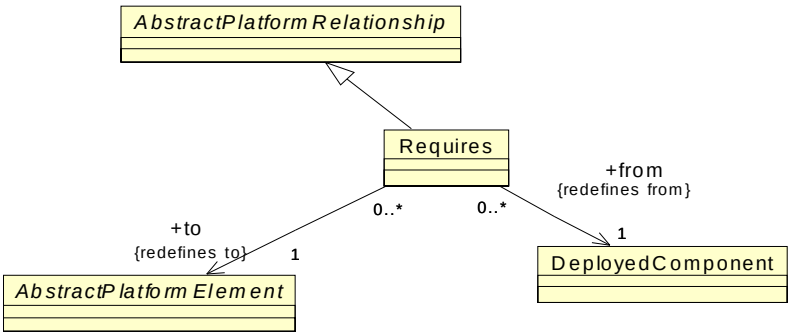

Figure 15.5 - ProvisioningRelations Class Diagram

15.7.1 Requires Class

Requires defines semantic relationship between a DeployedComponent and an AbstractPlatformElement.

Superclass

PlatformRelationship

Associations

from:DeployedComponent[1]	The DeployedComponent that is the source of the relationship (the from-endpoint).
to:AbstractPlatformElement[1]	The AbstractPlatformElement that is the target of the relationship (the to-endpoint).

Semantics

15.8 PlatformActions Class Diagram

The PlatformActions class diagram defines meta-model elements that represent specific actions that are the endpoints of certain Platform relationships. The elements of this diagram extend ActionElement from the KDM Action package.

The classes and associations of the PlatformActions diagram are shown in Figure 15.6.

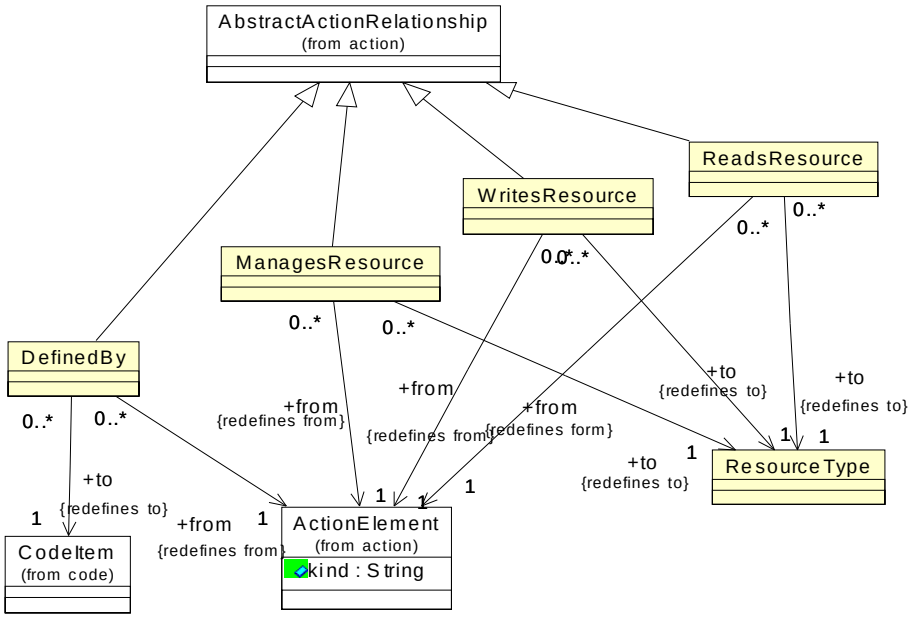

Figure 15.6 - PlatformActions Class Diagram

15.8.1 ManagesResource Class

ManagesResource class follows the pattern of a “resource action relationship.” It represents various types of accesses to platform resources that are not related to the flow of data to and from the resource. ManagesResource relationship is similar to Addresses relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the PlatformAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:ResourceType[1]	the resource being accessed

Constraints

1. This relationship should not be used in Code models.

15.8.2 ReadsResource Class

ReadsResource class follows the pattern of a “resource action relationship.” It represents various types of accesses to platform resources where there is a flow of data from the resource. ReadsResource relationship is similar to Reads relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the PlatformAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:ResourceType[1]	the resource being accessed

Constraints

1. This relationship should not be used in Code models

15.8.3 WritesResource Class

WritesResource class follows the pattern of a “resource action relationship.” It represents various types of accesses to platform resources where there is a flow of data to the resource. WritesResource relationship is similar to Writes relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the PlatformAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:ResourceType[1]	the resource being accessed

Constraints

1. This relationship should not be used in Code models.

15.8.4 DefinedBy Class

DefinedBy is a meta-model element that represents association between a platform resource and the logical package that describes the interface to this resource. The CodeItem at the to-endpoint of this KDM relationship is usually an interface or a package.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:CodeItem[1]	the CodeItem describing the resource

Constraints

1. This relationship cannot be used in the Code Model.

Semantics

DefinedBy is an optional relationship. The implementer shall correctly associate the platform resource with the corresponding logical definition of this resource (usually a Signature, an Interface, or a Package). The logical description of the package usually refers to some external implementation, as platform resources are usually described by some third-party packages, provided as part of the runtime platform of the application. Individual API calls corresponding to the given resource, should have the CompliesTo relations to the individual API descriptions the definition represented by the CodeItem at the to-endpoint of the DefinedBy relationship.

15.9 Deployment Class Diagram

The Deployment class diagram defines meta-model elements that represent deployment elements and their relations. In particular, the elements of the Deployment diagram address physical structures and how logical components are mapped to these physical structures.

The classes and associations of the Deployment diagram are shown in Figure 15.7.

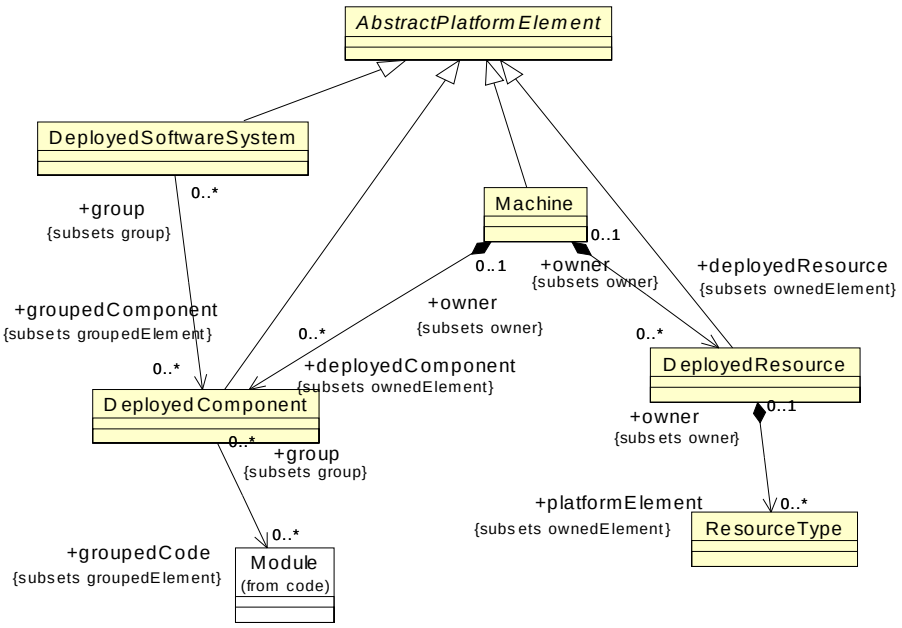

Figure 15.7 - Deployment Class Diagram

15.9.1 DeployedComponent Class

The DeployedComponent represents a unit of deployment as defined by a particular platform. Major platform parts provide a packaging mechanism of deploying application functionality. Deployment component is a replaceable unit of an application. For example, DLL, shared library, COM, Eclipse plugin, Executable, Jar file, War file for Tomcat, SQL Stored procedure, CORBA module, EJB, JavaBean, Jakarta Struts Action, Jakarta Struts Form.

Superclass

AbstractPlatformElement

Associations

groupedCode:Module[0..*]	The code components that are deployed to the target DeployedComponent (KDM grouping association).

Semantics

15.9.2 DeployedSoftwareSystem Class

The DeployedSoftwareSystem is a meta-model element that represents an instance of a software system at deployment or at the initialization time. DeployedSoftwareSystem is a physical instance of some logical SoftwareSystem. DeployedSoftwareSystem is associated with a set of DeployedComponents, which correspond to the set of logical components of the logical SoftwareSystem. Each SoftwareSystem involves one or more Components. Some components can be involved in more than one SoftwareSystem (allowing description of the so-called Software Product Lines). Each

Component involves one or more model Modules. Again, each Module can be involved in more than one Component. Component is a unit of deployment. Each logical component can be deployed multiple times, each time represented by a unique DeploymentComponent element. DeployedSoftwareSystem is a counterpart of the corresponding logical SoftwareSystem.

Superclass

AbstractPlatformElement

Associations

groupedComponent:DeployedComponent[0..*]	The set of physical DeployedComponents that make up the target system. The physical components correspond to the logical components of the system.

Semantics

15.9.3 Machine Class

The Machine is a meta-model element that represents the hardware node which hosts deployed components.

Superclass

AbstractPlatformElement

Associations

deployedComponent:DeployedComponent[0..*]	The set of DeployedComponent elements deployed to this node.
deployedResource:DeployedResource[0..*]	The set of DeployedResource elements deployed to this node.

Semantics

15.9.4 DeployedResource Class

The DeployedResource is a meta-model element that represents a set of platform resource instances as they are deployed in a particular deployment configuration. DeployedResource is associated with a set of ResourceType elements. DeployedResource provides a unique physical context for a logical resource, as each logical resource can be associated with multiple DeployedResources.

Superclass

AbstractPlatformElement

Associations

platformElement:ResourceType[0..*]	The set of ResourceTypes that are deployed into the target DeployedResource.

Semantics

15.10 RuntimeResources Class Diagram

The RuntimeResources class diagram defines meta-model elements that represent dynamic structures (instances of some logical entities and their relationships) that emerge at the so-called “run time” of the software system. For example, dynamic entities include processes and threads. Instances of processes and threads can be created dynamically and in many cases relations between the dynamically created instances of processes and threads are an essential part of the knowledge of existing systems. Another example of dynamic structures involves deployed components that are loaded dynamically.

The classes and associations of the RuntimeResources diagram are shown in Figure 15.8.

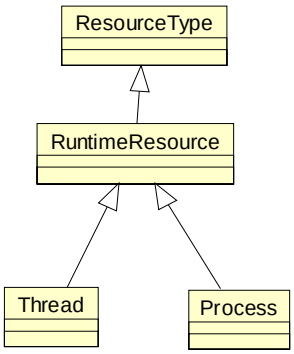

Figure 15.8 - RuntimeResources Class Diagram

15.10.1 RuntimeResource (generic)

The RuntimeResource is a generic meta-model element that represents an entity that has its own execution thread (for example, process or thread). RuntimeResource is subclassed by Process and Thread. In the meta-model RuntimeResource is used as the endpoint of certain relationships.

Superclass

ResourceType

Semantics

15.10.2 Process Class

The Process is a meta-model element that represents instances of processes.

Superclass

RuntimeResource

Semantics

15.10.3 Thread Class

The Thread is a meta-model element that represents instances of the so-called threads (light-weight processes).

Superclass

RuntimeResource

Semantics

15.11 RuntimeActions Class Diagram

The RuntimeActions class diagram defines meta-model elements that represent specific Runtime actions as the endpoints of certain Runtime relationships. The elements of this diagram extend ActionElement from the KDM Action package.

The classes and associations that make up the RuntimeActions diagram are shown in Figure 15.9.

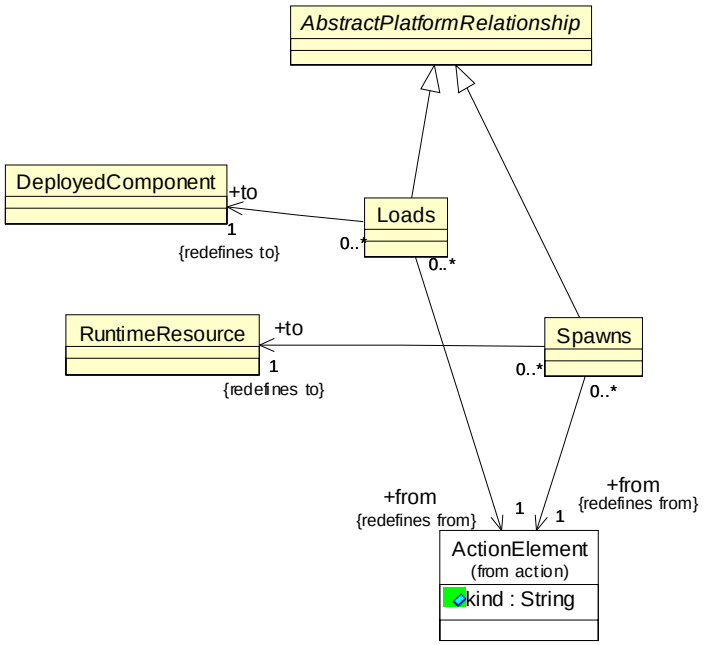

Figure 15.9 - RuntimeActions Class Diagram

15.11.1 Loads Class

The Loads class is a meta-model element that represents “dynamic loading relationship” between a LoadingService action endpoint and the DeployedComponent.

In the meta-model Loads is a subclass of a generic element RuntimeRelation.

Superclass

AbstractPlatformRelationship

Associations

from:ActionElement[1]	“abstracted” action element owned by some resource
to:DeploymentComponent[1]	The component that is being loaded.

Semantics

15.11.2 Spawns Class

The Spawns class is a meta-model element that represents “dynamic process creation” or “dynamic thread creation” relationship between a SpawningService action endpoint and the RunnableInterface (Process or Thread).

Superclass

AbstractPlatformRelationship

Associations

from:ActionElement[1]	“abstracted” action element owned by some resource
to:RuntimeResource[1]	The runtime resource element (Process or Thread) that is being spawned.

Semantics

15.12 ExtendedPlatformElements Class Diagram

The ExtendedPlatformElements class diagram defines two “wildcard” generic elements for the code model as determined by the KDM model pattern: a generic platform entity and a generic platform relationship.

The classes and associations of the ExtendedPlatformElements diagram are shown in Figure 15.10.

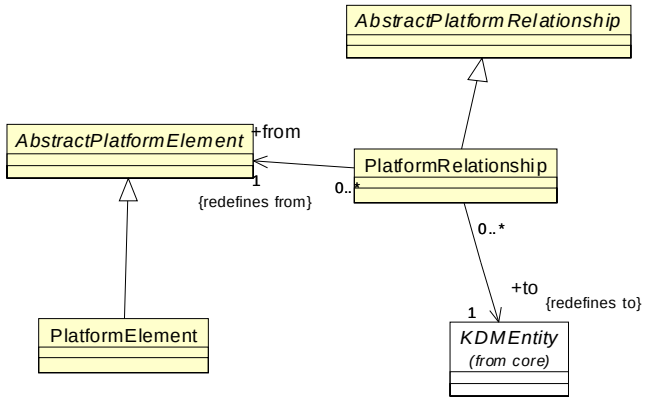

Figure 15.10 - ExtendedPlatformElements Class Diagram

15.12.1 PlatformElement Class (generic)

The PlatformElement class is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractPlatformElement

Constraints

1. PlatformElement should have at least one stereotype

Semantics

A platform entity with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity types of the platform model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

15.12.2 PlatformRelationship Class (generic)

The PlatformRelationship class is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractPlatformRelationship

Associations

from:AbstractPlatformElement[1]	the platform element endpoint
to:KDMEntity[1]	the target of the relationship

Constraints

1. PlatformRelationship should have at least one stereotype

Semantics

A platform relationship with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model relationship type of the platform model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

16 UI Package

16.1 Overview

The UI package defines a set of meta-model elements whose purpose is to represent facets of information related to user interfaces, including their composition, their sequence of operations, and their relationships to the existing software systems.

The UI package defines an *architectural viewpoint* for the UI domain.

- **Concerns**:
    - What are the distinct elements of the user interface of the systems?
    - What is the organization of the user interface?
    - How user interface uses artifacts of the system (for example, images) ?
    - What data flows originate from the user interface ?
    - What data flows output to the user interface?
    - What control flows are initiated by the user interface?
- **Viewpoint language:**

  UI views conform to KDM XMI schema. The *viewpoint language* for the UI *architectural viewpoint* is defined by the UI package. It includes an abstract entity AbstractUIElement, several generic entities, such as UIResource, UIDisplay, as well as several concrete entities, such as Screen, Report, UIField, UIAction, UIEvent, etc. The viewpoint language for the UI architectural viewpoint also includes several relationships, which are subclasses of AbstractUIRelationship.
- **Analytic methods**:

  The UI architectural viewpoint supports the following main kinds of checking:
    - Data flow (for example, what action elements read from a given UI element; what action elements write to a given UI element; what action elements manage a given UI element)
    - Control flow (for example, what action elements are triggered by events in a given UI element; what action elements operate on a given UI element)
    - Workflow (what UI elements will be displayed after the given one; what UI elements are displayed before the given one)

  UI Views are used in combination with Code views and Inventory views.
- **Construction methods:**
    - UI views that correspond to the KDM UI *architectural viewpoint* are usually constructed by analyzing Code views for the given system as well as the UI-specific configuration artifacts. The UI extractor tool uses the knowledge of the API and semantics for the given run-time platform to produce one or mode UI views as output
    - As an alternative, for some languages like Cobol, in which the elements of the UI are explicitly defined by the language, the UI views are produced by the parser-like tools which take artifacts of the system as the input and produce one or mode UI views as output (together with the corresponding Code views)

- Construction of the UI view is determined by the semantics of the UI platform, and it based on the mapping from the given UI platform to KDM; such mapping is specific only to the UI platform and not to a specific software system
- The mapping from a particular UI platform to KDM may produce additional information (system-specific, or platform-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations.

16.2 Organization of the UI Package

The UI package consists of the following 6 class diagrams.

1. UIModel
2. UIInheritances
3. UIResources
4. UIRelations
5. UIActions
6. ExtendedUIElements

The UI package depends on the following packages:

- Action
- Code
- kdm
- Source
- Core

16.3 UIModel Class Diagram

The UIModel class diagram follows the uniform pattern for KDM models to extend the KDM framework with specific meta-model elements related to static representations of the principal components of a user interface. The class diagram shown in Figure 16.1 captures these classes and their relations.

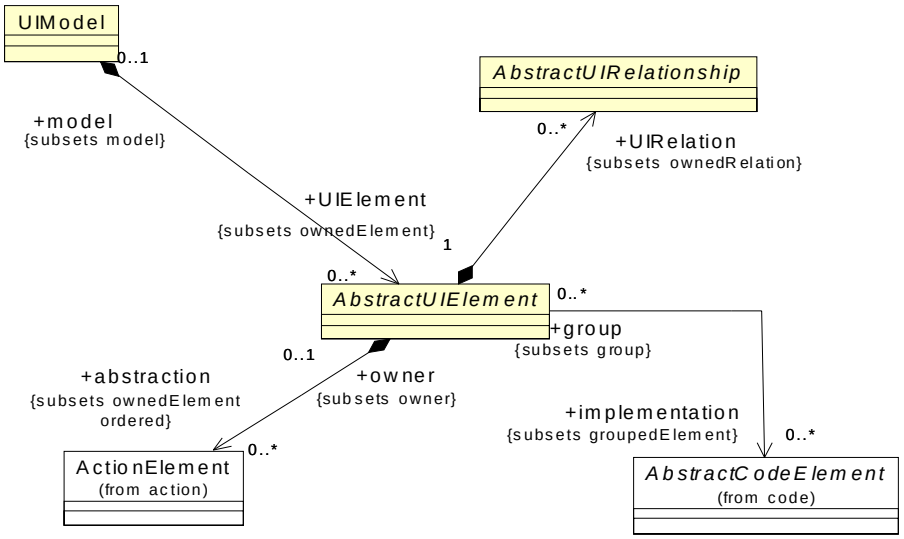

Figure 16.1 - UIModel Class Diagram

16.3.1 UIModel Class

The UIModel is the specific KDM model that corresponds to the user interface of the existing software system.

Superclass

KDMModel

Associations

UIElement:UIElement[0..*]	user interface elements owned by the given UIModel

Semantics

UIModel provides a container for various user-interface elements. The implementer shall arrange user-interface elements into one or more UIModel containers.

16.3.2 AbstractUIElement Class (abstract)

The AbstractUIElement is the abstract superclass for various concrete user interface elements. As such, it is the class that represents both compound and elementary items in a model of a system's user interface.

Superclass

KDMEntity

Associations

UIRelation:AbstractUIRelationship[0..*]	UI relationships originating from the given UI element
abstraction:ActionElement[0..*]	owned “abstraction” actions
implementation:AbstractCodeElement[0..*]	Grouped association to AbstractCodeElement from some CodeModel that are represented by the current UI element.
source: SourceRef[0..1]	link to the physical artifact for the given UI element

Constraints

1. Implementation AbstractCodeElement should be owned by some CodeModel.
2. Implementation AbstractCodeElement should be subclasses of ComputationalObject or ActionElement.
3. Abstraction ActionElement should be owned by the same UIModel.

Semantics

The implementer shall map specific user interface element types determined by the particular user-interface system of the existing software system, into concrete subclasses of the AbstractUIElement. The implementer shall map each user interface element into some instance of the AbstractUIElement. Implementation elements are one or more ComputationalObjects or ActionElements from some CodeModel that are represented by the current UI element. “Abstraction” actions may be used to represent precise semantics of the UI Element.

16.3.3 AbstractUIRelationship Class (abstract)

The AbstractUIRelationship is the abstract superclass for various user interface relationships.

Superclass

KDMRelationship

Semantics

The implementer shall map specific user interface association types determined by the particular user-interface system of the existing software system, into concrete subclasses of the AbstractUIRelationship. The implementer shall map each user interface association into some instance of the AbstactUIRelationship.

16.4 UIInheritances Class Diagram

The UIInheritances class diagram defines how classes of the UI package subclass core meta-model elements from the KDM Core package. The classes and associations that make up the UIInheritances class diagram are shown in Figure 16.2.

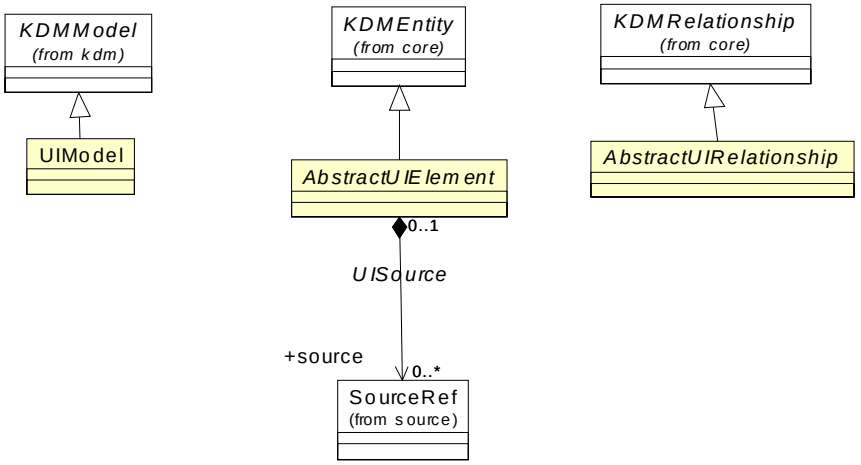

Figure 16.2 - UIInheritances Class Diagram

16.5 UIResources Class Diagram

The UIResource class diagram defines several specific KDM containers that own collections of user interface elements. The class diagram shown in Figure 16.3 captures these classes and their relations.

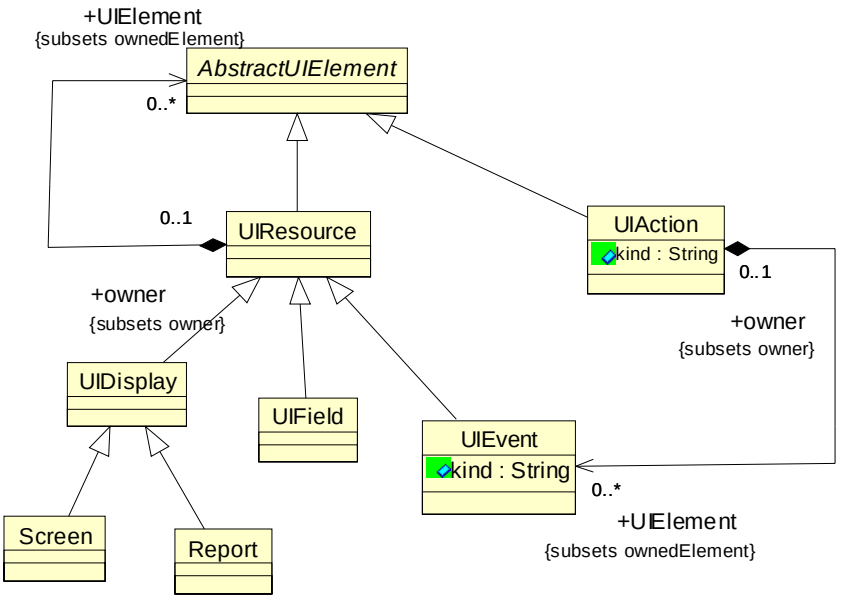

Figure 16.3 - UIResources Class Diagram

16.5.1 UIResource Class (generic)

The UIResource is the superclass for several user interface elements that can be containers for other user interface elements. For example, it represents a compound unit of display. This is a generic element.

Superclass

AbstractUIElement

Associations

UIElement:UIElement[0..*]	UI elements owned by this UIResource

Constraints

1. UIResource should have at least one stereotype.

Semantics

UIResource is a generic element with under specified semantics. It can be used as an *extension point*.

16.5.2 UIDisplay Class (generic)

The UIDisplay is the superclass of Screen and Report. It represents a compound unit of display.

Superclass

UIResource

Constraints

1. UIDisplay should have at least one stereotype.

Semantics

UIDisplay is a generic element with under specified semantics. It can be used as an *extension point*.

16.5.3 Screen Class

The Screen is a compound unit of display, such as a Web page or character-mode terminal that is used to present and capture information. The screen may be composed of multiple instances of AbstractUIElement and its subclasses.

Superclass

UIDisplay

Semantics

16.5.4 Report Class

The Report is a compound unit of display, such as a printed report, that is used to present information. The report may be composed of multiple instances of AbstractUIElement and its subclasses.

Superclass

UIDisplay

Semantics

16.5.5 UIField Class

The UIField is a unit of display, such as a control on a form, a text field on a character-mode terminal, or a field printed on a report.

Superclass

UIResource

Semantics

16.5.6 UIEvent Class

The UIEvent class is a meta-model element representing various events and callbacks associated with user interfaces. This class follows the KDM event pattern, common to Resource Layer packages.

Superclass

UIResource

Attributes

kind:String	represents the nature of the action performed by this Event

16.5.7 UIAction Class

UIAction class follows the pattern of a “resource action” class, specific to the UI package. The nature of the action represented by a particular element is designated by its “kind” attribute.

Superclass

AbstractUIElement

Attributes

kind:String	represents the nature of the action performed by this element

Associations

UIElement:UIEvent[0..*]	UI events owned by this UIAction

Semantics

16.6 UIRelations Class Diagram

The UIRelations class diagram provides basic meta-model constructs to define the binding between elements of a display and their content. The class diagram shown in Figure 16.4 captures these classes and their relations.

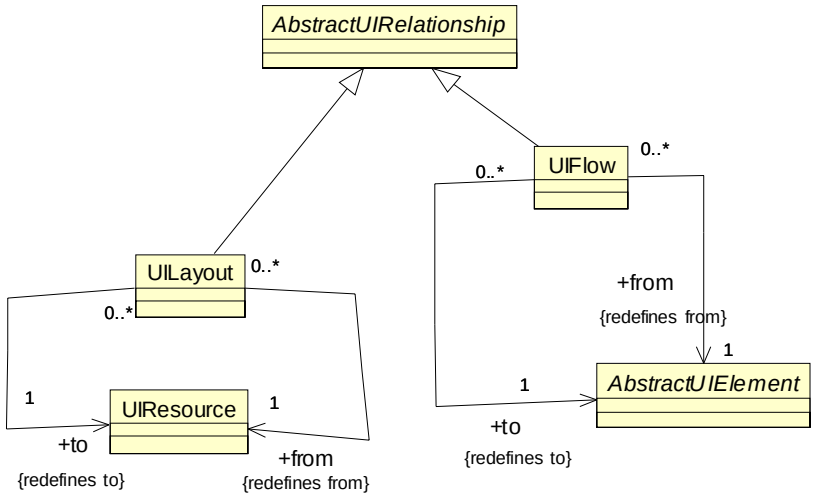

Figure 16.4 - UIRelations Class Diagram

16.6.1 UIFlow Class

The UIFlow relationship class captures the behavior of the user interface as the sequential flow from one instance of Display to another.

Superclass

AbstractUIRelationship

Associations

from:AbstractUIElement[1]

to:AbstractUIElement[1]

Semantics

16.6.2 UILayout Class

The UILayout relationship class captures an association between two instances of Display – one that defines the content for a portion of a user interface, and one that defines its layout.

Superclass

AbstractUIRelationship

Associations

from:UIResource[1]	the origin UI Resource
to:UIResource[1]	the target UI Resource

Semantics

16.7 UIActions Class Diagram

The UIActions class diagram defines several KDM relations for the UI package. It provides basic meta-model constructs to define the sequence of display in a user interface, and the mapping between a user interface and the events it may generate.

The class diagram shown in Figure 16.5 captures these classes and their relations.

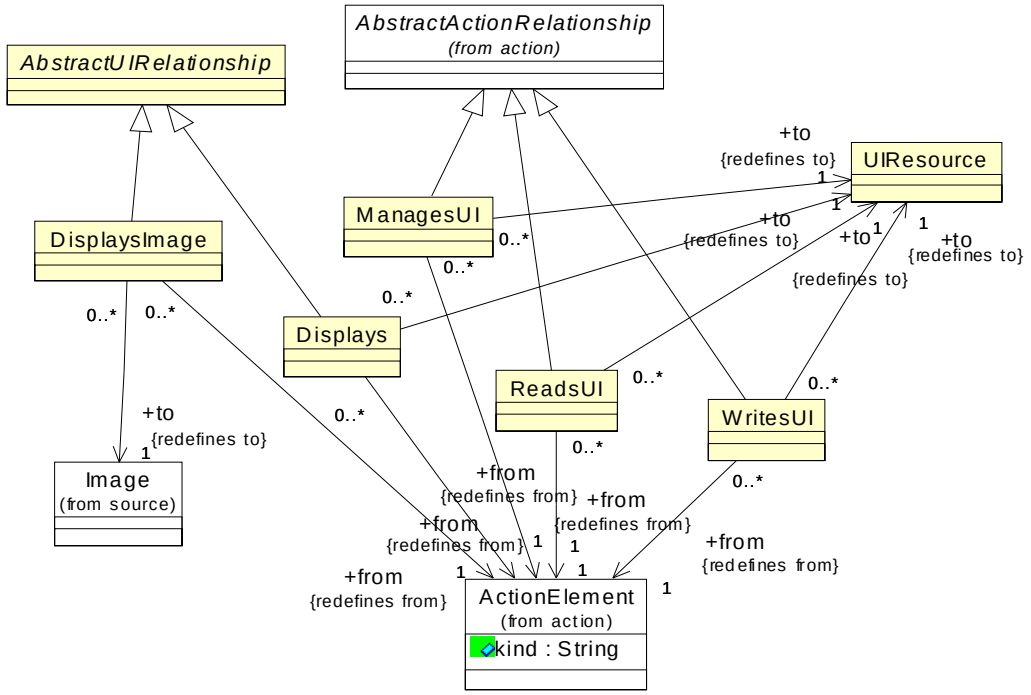

Figure 16.5 - UIActions Class Diagram

16.7.1 Displays Class

The Displays relationship class represents the relationship between an instance of CallableInterface and the instance of UIElement that is presented on the interface as a result of the execution of the CallableInterface.

Superclass

AbstractUIRelationship

Associations

from:ActionElement[1]	the ActionElement that displays a certain UI resource
to:UIResource[1]	the target UI resource

Semantics

16.7.2 DisplaysImage Class

The DisplaysImage captures the relationship between an image file – an instance of Image – and its presentation on a user interface – an instance of DisplayUnit.

Superclass

AbstractUIRelationship

Associations

from:ActionElement[1]	The ActionElement that displays a certain Image.
to:Image[1]	the target Image element

Semantics

16.7.3 ManagesUI Class

ManagesUI class follows the pattern of a “resource action relationship.” It represents various types of accesses to user interface resources that are not related to the flow of data to and from the resource. ManagesUI relationship is similar to Addresses relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the UIAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:UIResource[1]	the user interface resource being accessed

Constraints

1. This relationship should not be used in Code models.

16.7.4 ReadsUI Class

ReadsUI class follows the pattern of a “resource action relationship.” It represents various types of accesses to user interface resources where there is a flow of data from the resource. ReadsUI relationship is similar to Reads relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the UIAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:UIResource[1]	the user interface resource being accessed

Constraints

1. This relationship should not be used in Code models.

16.7.5 WritesUI Class

WritesUI class follows the pattern of a “resource action relationship.” It represents various types of accesses to user interface resources where there is a flow of data to the resource. WritesUI relationship is similar to Writes relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the UIAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:UIResource[1]	the user interface resource being accessed

Constraints

1. This relationship should not be used in Code models.

16.8 ExtendedUIElements Class Diagram

The ExtendedUIElements class diagram defines two “wildcard” generic elements for the UI model as determined by the KDM model pattern: a generic UI entity and a generic UI relationship.

The class diagram shown in Figure 16.6 captures these classes and their relations.

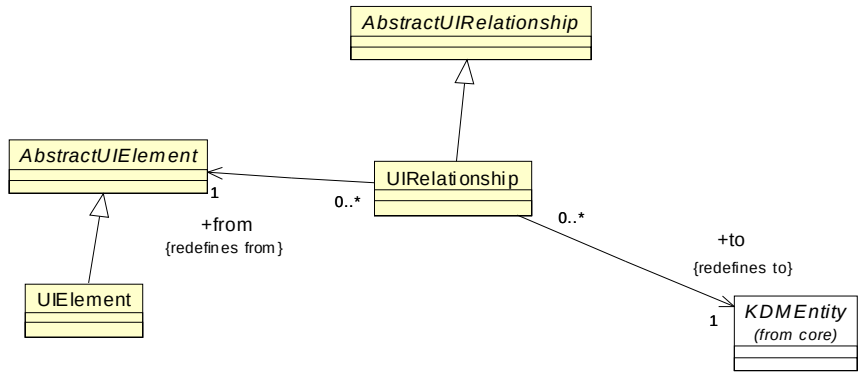

Figure 16.6 - ExtendedUIElements Class Diagram

16.8.1 UIElement Class (generic)

The UIElement class is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractUIElement

Constraints

1. UIElement should have at least one stereotype.

Semantics

A UI entity with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity type of the UI model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

16.8.2 UIRelationship Class (generic)

The UIRelationship relationship is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractUIRelationship

Associations

from:AbstractUIElement[1]	the origin UI element
to:KDMEntity[1]	the target KDM entity

Constraints

1. UIRelationship should have at least one stereotype.

Semantics

A UI relationship with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model relationship type of the UI model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

17 Event Package

17.1 Overview

The Event package defines a set of meta-model elements whose purpose is to represent high-level behavior of applications, in particular event-driven state transitions. Elements of the KDM Event package represent states, transitions, and event. States can be concrete, for example, the ones that are explicitly supported by some state-machine based runtime framework or a high level programming language, such as CHILL. On the other hand, KDM Event model can represent abstract states, for example, states that are associated with a particular algorithm, resource, or a user interface.

The Event packages defines an *architectural viewpoint* for the Event domain.

- **Concerns**
  - What are the distinct states involved in the behavior of the software system?
  - What are the events that cause transitions between states?
  - What action elements are executed in a given state?
- **Viewpoint language:**

  Event views conform to KDM XMI schema. The *viewpoint language* for the Event *architectural viewpoint* is defined by the Event package. It includes an abstract entity AbstractEventElement, generic entity EventResource, UIDisplay, as well as several concrete entities, such as State, Transition, Event, EventAction, etc. The viewpoint language for the UI architectural viewpoint also includes several relationships, which are subclasses of AbstractEventRelationship.
- **Analytic methods**:

  The Event architectural viewpoint supports the following main kinds of checking:
  - Reachability (for example, what states are reachable from the given state).
  - Control flow (for example, what action elements are triggered by a given state transition; what action elements will be executed for a given traversal of the state transition graph).
  - Data flow (what data sequences correspond to a given traversal of the state transition graph).

  Event Views are used in combination with Code views, Data views, Platform views and Inventory views.
- **Construction methods**:
  - Event views that correspond to the KDM Event architectural viewpoint are usually constructed by analyzing Code views for the given system as well as the configuration artefacts specific to the event-driven framework. The Event extractor tool uses the knowledge of the API and semantics of the event-driven framework to produce one or mode Event views as output.
  - Construction of the Event view is determined by the semantics of the event-driven framework, and it based on the mapping from the given event-driven framework to KDM; such mapping is specific only to the event-driven framework and not to a specific software system.
  - The mapping from a particular event-driven framework to KDM may produce additional information (system-specific, or platform-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations.

17.2 Organization of the Event Package

The Event package consists of the following 6 class diagrams.

1. EventModel
2. EventInheritances
3. EventResources
4. EventRelations
5. EventActions
6. ExtendedEventElements

The Event package depends on the following packages:

- Core
- kdm
- Source
- Code
- Action

17.3 EventModel Class Diagram

The EventModel class diagram follows a uniform pattern for KDM models to extend the KDM framework with specific meta-model elements related to event-driven state transition behavior. The class diagram shown in Figure 17.1 captures these classes and their relations.

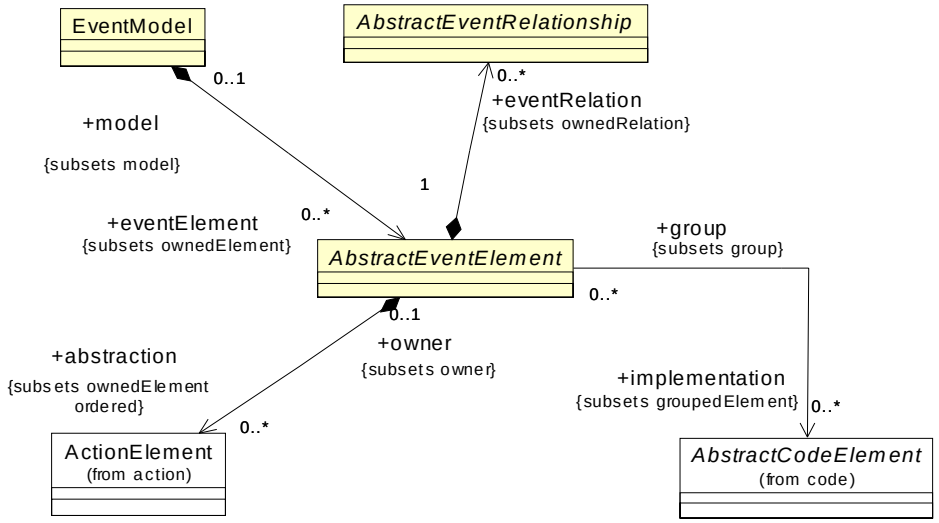

Figure 17.1 - EventModel Class Diagram

17.3.1 EventModel Class

The EventModel is a specific KDM model that represents entities and relations describing events and responses to events in an enterprise application.

Superclass

KDMModel

Associations

eventElement:AbstractEventElement[0..*]	event elements owned by the given event model

Semantics

EventModel is a container for instances of event elements. The implementer shall arrange event elements into one or more event models.

17.3.2 AbstractEventElement Class (abstract)

The AbstractEventElement is an abstract superclass for various event elements.

Superclass

KDMEntity

Associations

eventRelation:AbstractEventRelationship[0..*]	event relations owned by the given element
abstraction:ActionElement[0..*]	owned “abstracted” action elements
implementation:AbstractCodeElement[0..*]	Group association to AbstractCodeElement elements from some CodeModel that are represented by the current EventElement.
source:SourceRef[0..*]	traceability links to the “source code” of the artifact

Constraints

1. Implementation AbstractCodeElement should be owned by some CodeModel.
2. Implementation AbstractCodeElement should be subclass of ComputationalObject or ActionElement.
3. Abstraction ActionElement should be owned by the same EventModel.

Semantics

Implementation AbstractCodeElements are one or more ComputationalObjects or ActionElements that are represented by the current EventElement. “Abstraction” actions can be used to represent precise semantics of the EventElement.

17.3.3 AbstractEventRelationship Class (abstract)

The AbstractEventRelationship is the superclass of associations of the event model. This is an abstract meta-model element for representing various relations involving states and events.

Superclass

KDMRelationship

Semantics

17.4 EventInheritances Class Diagram

The EventInheritances class diagram defines how classes of the Event package inherit core meta-model classes from KDM Core package. The classes and associations that make up the EventInheritances diagram are shown in Figure 17.2.

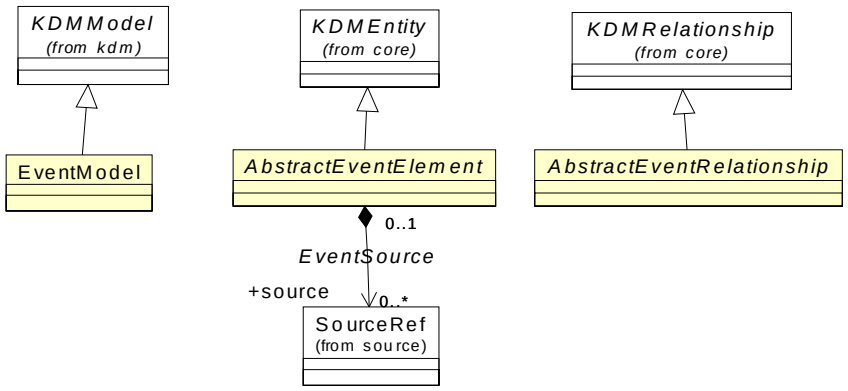

Figure 17.2 - EventInheritances Class Diagram

17.5 EventResources Class Diagram

The EventResources class diagram defines specific event elements. The class diagram shown in Figure 17.3 captures these classes and their relations.

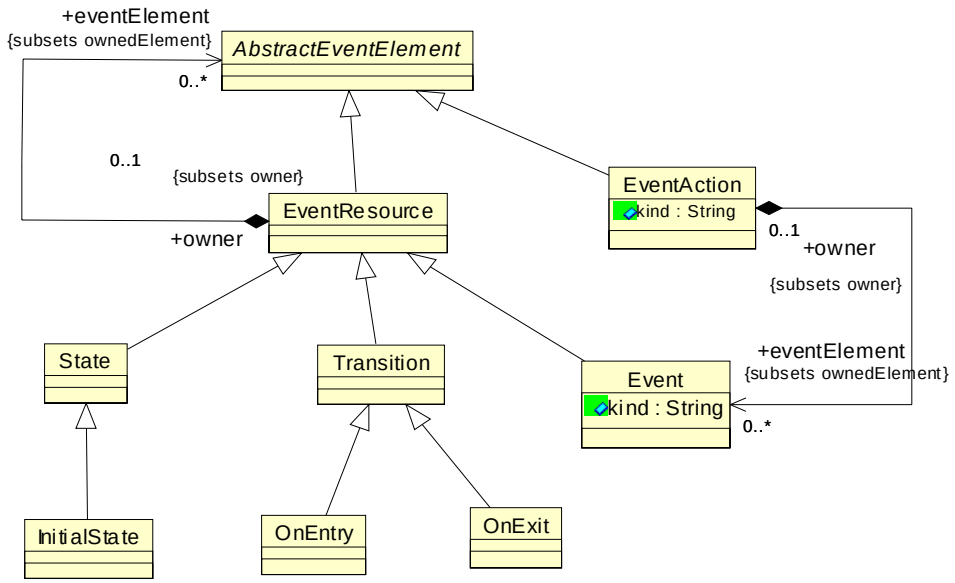

Figure 17.3 - EventResourcesClass Diagram

17.5.1 EventResource Class (generic)

The EventResource is the generic AbstractEventElement that can be instantiated in KDM instances.

Superclass

AbstractEventElement

Associations

eventElement:AbstractEventElement[0..*]	Event elements owned by this EventResource

Semantics

17.5.2 Event Class

The Event is the generic AbstractEventElement that can be instantiated in KDM instances.

Superclass

EventResource

Attributes

kind:String	represents the nature of this Event

Semantics

17.5.3 State Class

The State class represents a state associated with certain behavior. This can be a concrete state, for example, supported by a state-machine runtime framework. This can also be an abstract state associated with some process, algorithm, component, or resource discovered during the analysis of the software system. An example of an abstract state is the step of the protocol that involves a messaging resource. An abstract state may not have any direct and tangible manifestation in the artifacts of the software system. On the other hand, a concrete state may be implemented in a tangible way, for example, using a variable or as a class provided by the application framework. States can be nested.

Superclass

EventResource

17.5.4 InitialState Class

The InitialState class is a subclass of the State class. It represents a default initial state.

Superclass

State

17.5.5 Transition Class

The Transition class represents a transition that is performed when a certain event is consumed is a certain state. Transition element should be owned by some state element. Transition can be associated with the corresponding Event by using the “ConsumesEvent” resource relation. A transition element can also own some Event elements. Transition does not have an “implementation” group. Instead, it is considered as some sort of a trigger. The association between the transition and corresponding behavior is achieved through the “abstraction” action container of the transition. Usually, this is a Calls action relation. For more complex situations, the “CodeGroup” capability of the “abstraction” action element can be used.

Superclass

EventResource

17.5.6 OnEntry Class

The OnEntry class represents specific transitions that are configured to be performed by the runtime framework when a certain state has been entered.

Superclass

Transition

17.5.7 OnExit Class

The OnExit class represents specific transitions that are configured to be performed by the runtime framework when a certain state has been

Superclass

Transition

17.5.8 EventAction Class

EventAction class follows the pattern of a “resource action” class, specific to the event package. The nature of the action represented by a particular element is designated by its “kind” attribute. Descriptions of the common platform action kind are provided in Part 3: Resource Layer actions.

Superclass

AbstractEventElement

Attributes

kind:String	represents the nature of the action performed by this element

Associations

eventElement:Event[0..*]	The set of Event elements that is owned by the current EventAction element.

17.6 EventRelations Class Diagram

EventRelations diagram defines meta-model relationship elements that represent several structural properties of event-driven systems. The class diagram shown in Figure 17.4 captures these classes and their relations.

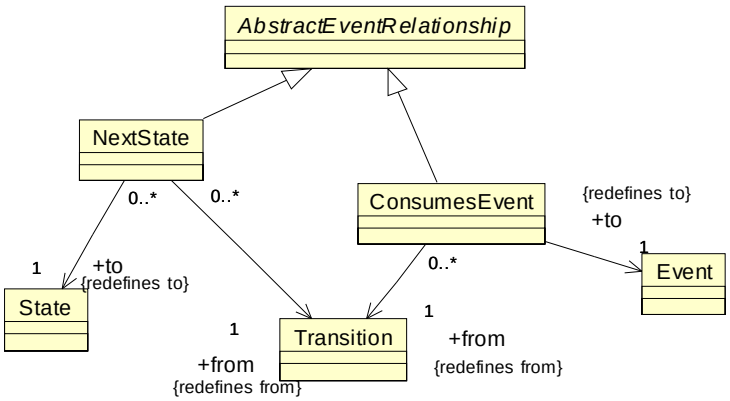

Figure 17.4 - EventRelations Class Diagram

17.6.1 NextState Class

The NextState class represents the knowledge that upon completion of the behavior associated with a certain transition element, the corresponding behavior will enter the given state. For example, in statically configured state-machine based frameworks this information can be derived from the initialization of framework specific data structures. When there

exists several NextState relations originating from a given transition, this means that an unspecified choice is made by the behavior associated with the transition. More precise “abstraction” can be provided by using the “abstraction” action containers associated with various elements involved.

Superclass

AbstractEventRelationship

Associations

to:Transition[1]	the transition
from:State[1]	the state

17.6.2 ConsumesEvent Class

The ConsumesEvent class represents the knowledge that a certain transition element is associated with a certain event. For example, in statically configured state-machine based frameworks this information can be derived from the initialization of framework specific data structures.

Superclass

AbstractEventRelationship

Associations

from:Transition[1]	the transition
to:Event[1]	the event

17.7 EventActions Class Diagram

The EventActions class diagram defines basic KDM relations between EventActions and other entities from the Event package. The class diagram shown in Figure 17.5 captures these classes and their relations.

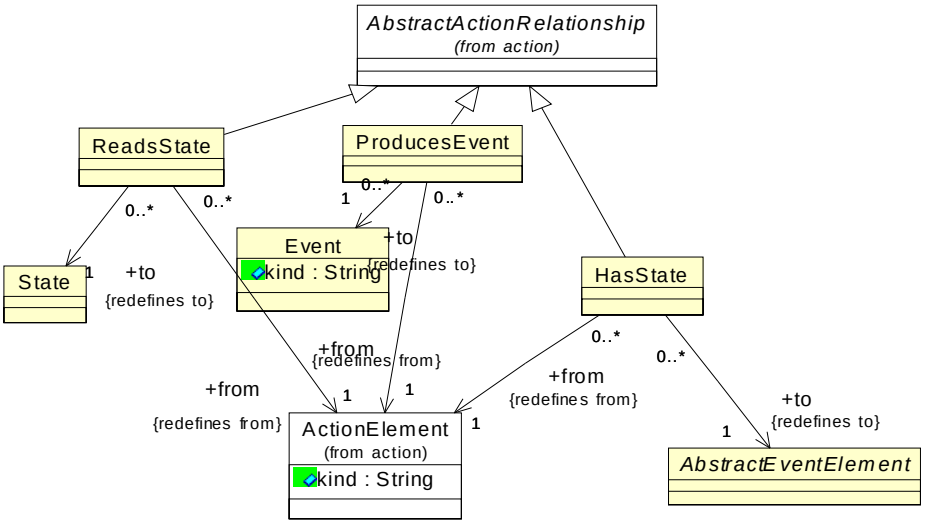

Figure 17.5 - EventActions Class Diagram

17.7.1 ReadsState Class

ReadsState class follows the pattern of a “resource action relationship.” It represents various types of accesses to the state-based runtime framework that provides a concrete implementation of states, where access is made to a particular state (for example, accessing the current state, setting the next state). ReadsState relationship is similar to Addresses relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the EventAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:EventResource[1]	the event resource being accessed

Constraints:

1. This relationship should not be used in Code models.
2. The to endpoint of the relationship should be State of one of its subclasses.

17.7.2 ProducesEvent Class

ProducesEvent class follows the pattern of a “resource action relationship.” It represents various types of accesses to the state-based runtime framework where the application produces the event. ProducesEvent relationship is similar to Writes relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the EventAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:EventResource[1]	the event resource being produced

Constraints

1. This relationship should not be used in Code models.
2. The “to” endpoint of the relationship should be Event.

17.7.3 HasState Class

HasState class follows the pattern of a “resource action relationship.” HasState is a structural relationship. It does not represent resource manipulations. HasState relationship uses the “abstracted” action container mechanism to provide certain capabilities to other Resource Layer packages. “HasState” relationship makes it possible to associate an element of an event model with any resource.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:EventResource[1]	the event resource being accessed

Constraints

1. This relationship should not be used in Code models.

17.8 ExtendedEventElements Class Diagram

The ExtendedEventElements class diagram defines two “wildcard” generic elements for the event model as determined by the KDM model pattern: a generic event entity and a generic event relationship.

The classes and associations of the ExtendedEventElements diagram are shown in Figure 17.6.

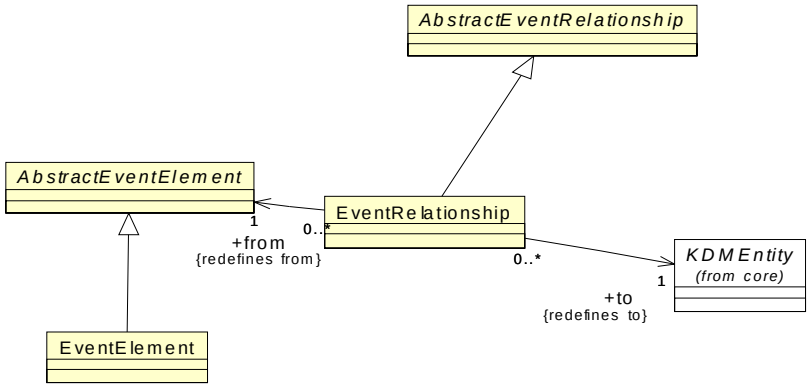

Figure 17.6 - ExtendedEventElements Class Diagram

17.8.1 EventElement Class (generic)

The EventElement class is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractEventElement

Constraints

1. EventElement should have at least one stereotype.

Semantics

An event entity with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model entity type of the event model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

17.8.2 EventRelationship Class (generic)

The EventRelationship class is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractEventRelationship

Associations

from:AbstractEventElement[1]	the event element origin endpoint of the relationship
to:KDMEntity[1]	the target of the relationship

Constraints

1. EventRelationship should have at least one stereotype.

Semantics

An event relationship with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model relationship type of the event model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

18 Data Package

18.1 Overview

The KDM Data Package defines a set of meta-model elements whose purpose is to represent organization of data in the existing software system. Facts in the Data domain are usually determined by a Data Description Language (for example, SQL) but may in some cases be determined by the code elements. KDM Data model uses the foundation provided by the Code package related to the representations of simple datatypes. KDM Data model represents complex data repositories, such as record files, relational databases, structured data stream, XML schemas and documents.

The KDM Data package defines an *architectural viewpoint* for the Data domain.

- **Concerns**
  - What is the organization of persistent data in the software systems?
  - What are the information model supported by the software system?
  - What action elements read persistent data?
  - What action elements write persistent data?
  - What control flows are determined by the events corresponding to persistent data?
- **Viewpoint language**

  Data views conform to KDM XMI schema The viewpoint language for the Data architectural viewpoint is defined by the Data package. It includes abstract entities AbstractDataElement, AbstractContentElement, generic entities DataResource, DataContainer, ContentItem, as well as several concrete entities, such as Catalog, RelationalSchema, DataEvent, DataAction, ColumnSet, RecordFile, XMLSchema, etc. The viewpoint language for the Data architectural viewpoint also includes several relationships, which are subclasses of AbstractDataRelationship.
- **Analytic methods**:

  The Data architectural viewpoint supports the following main kinds of checking:
  - Data aggregation (the set of data items accessible from the given ColumnSet by adding data items through foreign key relationships to other tables).

  Data Views are used in combination with Code views and Inventory views.
- Construction methods:
  - Data views that correspond to the KDM Data architectural viewpoint are usually constructed by analyzing Data Definition Language artifacts for the given data management platform. The Data extractor tool uses the knowledge of the data management platform to produce one or mode Data views as output.
  - As an alternative, for some languages like Cobol, in which some elements of the Data are explicitly defined by the language, the Data views are produced by the parser-like tools which take artifacts of the system as the input and produce one or mode Data views as output (together with the corresponding Code views).
  - Construction of the Data view is determined by the semantics of the data management platform, and it based on the mapping from the given data management platform to KDM; such mapping is specific only to the data management platform and not to a specific software system.

- The mapping from a particular data management platform to KDM may produce additional information (system-specific, or platform-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations.

18.2 Organization of the Data Package

The Data package consists of the following 11 class diagrams:

1. Data Model
2. Data Inheritance
3. RelationalData
4. ColumnSet
5. StructuredData
6. ContentElements
7. ContentRelations
8. KeyIndex
9. KeyRelations
10. DataActions
11. ExtendedDataElements

The Data Package depends on the following packages:

- Core
- kdm
- Source
- Code
- Action

18.3 Data Model Class Diagram

The Data Model follows the uniform pattern for KDM models to extend the KDM Framework with specific meta-model elements related to data organization in complex data repositories. Figure 18.1 shows the classes and associations of the DataModel class diagram.

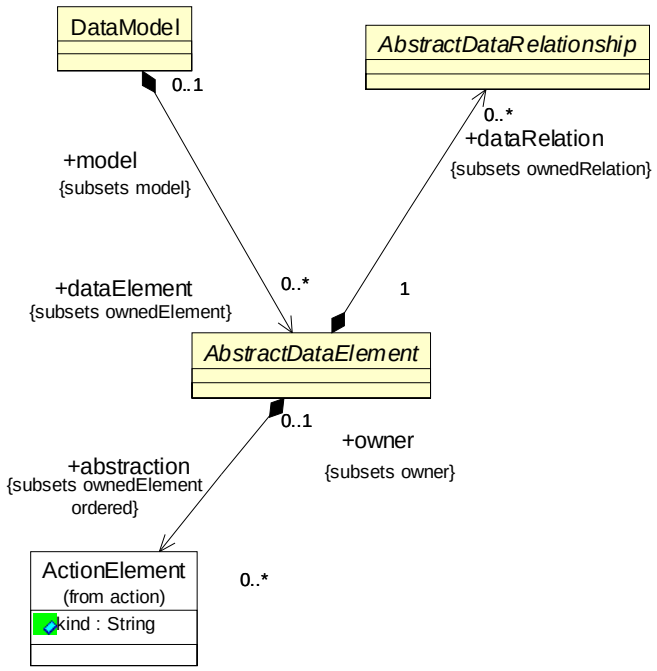

Figure 18.1 - Data Model

18.3.1 DataModel Class

The DataModel Class is the specific KDM model that corresponds to the logical organization of data of the existing software system, in particular, related to persistent data. DataModel follows the uniform pattern for KDM models.

Superclass

KDMModel

Associations

dataElement :DataElement[0..*]	data elements owned by the given DataModel

Semantics

Data model is a logical container for the instances of data elements. The implementer shall arrange the instances of the data elements into one or more DataModels.

18.3.2 AbstractDataElement Class (abstract)

The AbstractDataElement class is an abstract meta-model element that represents the discreet instance of a given data element within a system. For example, a *Customer_Number* is one type of data element that might be found within a system. Data model defines several specific subclasses of AbstractDataElement, corresponding to common subcategories of data elements.

Superclass

KDMEntity

Associations

abstraction: ActionElement[1]	the “abstracted” actions that are owned by the current element
dataRelation:DataRelation[0..*]	data relationships that originate from this data element
source: SourceRef[0..1]	link to the physical artifact for the given data element

Semantics

Abstracted actions are owned by the data model. Usually they provide an abstracted representation of one or more API calls in the code model. Abstracted actions own action relations to elements of code model, as well as some data relations.

Abstracted actions are ordered. The first action is the entry point.

18.3.3 AbstractDataRelationship Class (abstract)

An AbstractDataRelationship class is an abstract superclass of the meta-model elements that represent associations between data elements.

Superclass

KDMRelationship

Semantics

AbstractDataRelationship is an abstract class that is used to constrain the subclasses of KDMRelationship in the Data model.

18.4 Data Inheritances Class Diagram

The DataInheritances Diagram in Figure 18.2 shows how various data classes derive from the Core KDM classes. Each of the Data Package classes within this diagram inherits certain properties from KDM classes defined within the KDM Core Package.

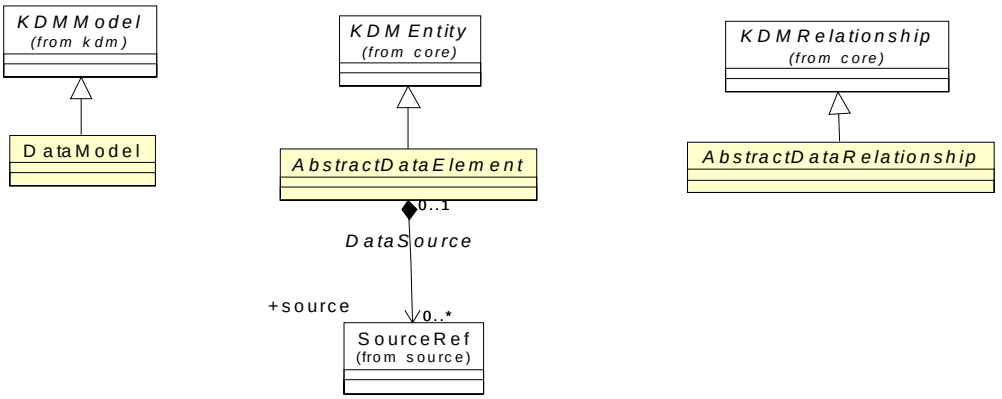

Figure 18.2 - DataInheritances Diagram

18.5 DataResources Class Diagram

The DataResources class diagram provides basic meta-model constructs to represent data elements within the KDM framework. The class diagram shown in Figure 18.3 captures these classes and their relations.

The DataResources diagram defines the framework for various data models. This framework follows the common pattern of the Runtme Resource Layer. Data model defines a generic DataResource meta-model element that represents various resources common to databases, such as a DataEvent and an IndexElement. Data model also defines a generic DataContainer class that represents various data containers, such as a relational schema, a database catalog, an XML schema, and a ColumnSet. DataContainer is a subclass of DataResource. DataContainer owns certain Data resources. Data model includes AbstractDataContent element which is a direct subclass of AbstractDataElement, and not a subclass of DataResource. Subclasses of AbstractContentElement are owned by XMLSchema element.

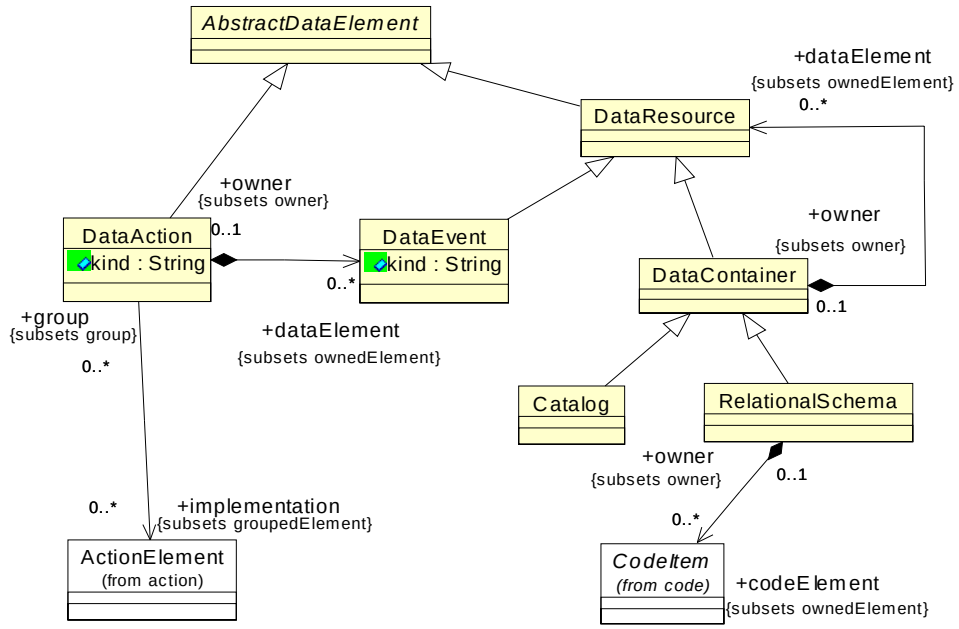

Figure 18.3 - RelationalData Class Diagram

18.5.1 DataResource Class (generic)

The DataResource class is a generic meta-model element that represents various database resources, such as DataEvent and IndexElement.

Superclass

AbstractDataElement

Constraints

1. DataResource should have at least one stereotype

Semantics

DataResource is a generic meta-model element with under specified semantics. DataResource is a database element that is associated with a certain data container, such as a Schema or a Table. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity type of the data model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation. DataResource is more specific than a generic ExtendedDataElement.

18.5.2 DataContainer Class (generic)

The DataContainer class is a generic meta-model element that represents various database containers.

Superclass

DataResource

Associations

dataElement :DataResource[0..*]	owned data resources

Semantics

DataContainer is a generic meta-model element with under specified semantics. DataContainer is a database element that is a logical container for Data resource, such as DataEvent or IndexElement. It is a concrete class that can be used as the base element of a new "virtual" meta-model entity type of the data model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation. DataContainer is more specific than a generic ExtendedDataElement.

18.5.3 Catalog Class

The Catalog class is the top level container that represents a relational or a hierarchical database.

Superclass

DataContainer

Semantics

18.5.4 RelationalSchema Class

The RelationalSchema class is a relational database schema.

Superclass

DataContainer

Associations

codeElement:CodeItem[0..*]	Stored procedures owned by this schema.

Semantics

Owned CodeElement represents stored procedures as well as scripts in data description language and data manipulation language, such as T-SQL. Data manipulation performed by embedded data manipulation statements (for example, embedded SQL) from code in some programming language (for example, Cobol or C) is represented via "abstracted data actions." Abstracted actions represent a "virtual" data manipulation statement, which is being implemented through embedded data manipulation constructs (and the corresponding "generated" API calls).

In the situation of the data manipulation and data description scripts that are executed directly by the relational database engine, KDM allows more tight integration of the corresponding CodeItem with the Data Model.

18.5.5 DataEvent Class

The DataEvent class is a meta-model element that represents various events in databases that can trigger execution of stored procedures, the so-called triggers. KDM models database events as "first class citizens" of the KDM representation.

Superclass

DataResource

Attributes

kind :String	semantic description of the data event

Semantics

Events are changes in entities or in relations among entities, so that the core KDM elements are entities and relations rather than events. However, KDM represents events as "first class citizens," although events might have to take on some of the character of entities for this to be acceptable. KDM data event represents various events in databases as KDM entities. Data events are associated with *triggers.* A *trigger* is a special kind of stored procedure that automatically executes when an event occurs in the database server. DML triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. DDL triggers execute in response to a variety of data definition language (DDL) events. These events primarily correspond to CREATE, ALTER, and DROP statements, and certain system stored procedures that perform DDL-like operations. Logon triggers fire in response to the LOGON event that is raised when a user session is being established.

As a subclass of AbstractDataElement, a DataEvent can own "abstracted" action element. Trigger is a stored procedure, which is represented as a CallableUnit, owned by a certain RelationalSchema. Trigger is associated with a data event through a Calls relationship, owned by the "abstracted" action of the corresponding data event. DataEvent is owned by a certain DataContainer.

18.5.6 DataAction Class

DataAction class follows the pattern of a "resource action" class, specific to the data package. The nature of the action represented by a particular element is designated by its "kind" attribute. Descriptions of the common platform action kind are provided in Part 3: Resource Layer actions.

Superclass

AbstractDataElement

Attributes

kind:String	represents the nature of the action performed by this element

Associations

implementation:ActionElement[0..*]	group association to ActionElement represented by the current DataAction
dataElement:DataEvent[0..*]	event elements owned by the current DataAction

Semantics

DataAction represents a "virtual" action element that represents the logical action performed by the runtime platform of the existing software system.

18.6 ColumnSet Class Diagram

The ColumnSet class diagram provides basic meta-model elements to define the tables and views of relational databases, segments of hierarchical databases, and record files as collections of columns. The class diagram shown in Figure 18.4 captures these classes and their relations.

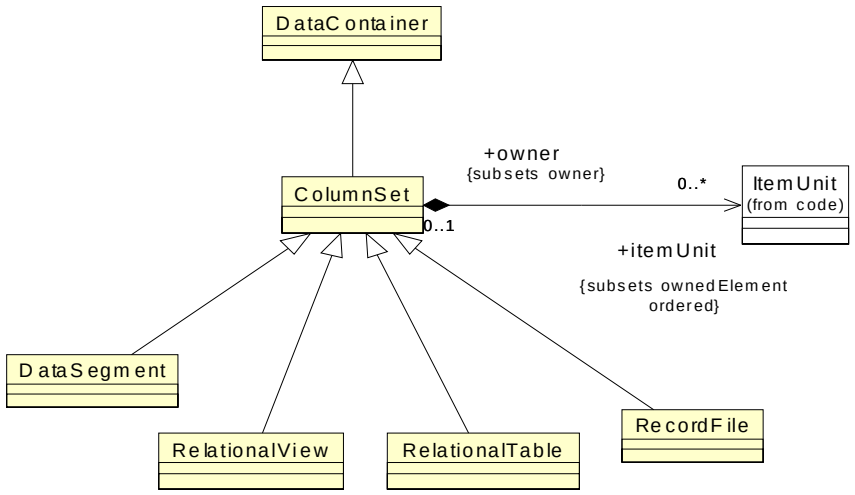

Figure 18.4 - ColumnSet Class Diagram

18.6.1 ColumnSet (generic)

The ColumnSet class is a generic meta-model element that represents collections of columns (also referred to as *fields*). Columns are modeled as ItemUnits.

Superclass

DataContainer

Associations

itemUnit :ItemUnit[0..*]	Individual columns owned by this ColumnSet are represented as data elements

Semantics

ColumnSet corresponds to an ISO/IEC 11404 Table datatype, whose values are collections of values in the product space of one or more *field* datatypes, such that each value in the product space represents an association among the values of the *fields*. Although the field datatypes may be infinite, any given value of a table datatype contains a finite number of associations.

KDM defines several concrete subclasses of ColumnSet to represent several common data organizations, such as relational Tables and Views, Record files and Segments of hierarchical databases.

Fields of the Columnset are represented as ItemUnits.

18.6.2 RelationalTable Class

A RelationalTable is a specific subclass of ColumnSet class that represents tables of relational databases.

Superclass

ColumnSet

Semantics

Tables are entities that contain all the data in relational databases. Each table represents a type of data that is meaningful to its users. A table definition is a collection of columns. In tables, data is organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field within the record. For example, a table that contains employee data for a company can contain a row for each employee and columns representing employee information such as employee number, name, address, job title, and home telephone number.

Tables in a relational database have the following main components:

- Columns. Each column represents some attribute of the object modeled by the table, such as a parts table having columns for ID, color, and weight.
- Rows. Each row represents an individual occurrence of the object modeled by the table. For example, the parts table would have one row for each part carried by the company.

The PlatformResource that corresponds to RelationalTable is DataManager.

Example (T-SQL)

```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn date,
           PRIMARY KEY(contract, recognizedOn))

CREATE PROCEDURE INSERT_RECOGNITION
(IN contractID int, IN amount decimal, IN recognizedOn date, OUT result int)
LANGUAGE SQL
BEGIN
        INSERT INTO revenueRecognitions VALUES( contractID, amount, recognizedOn);
        SET result = 1;
END

CREATE TRIGGER reminder1
ON Contracts.revenueRecognitions
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Sales', 16, 10)
GO

<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
        xmlns:platform="http://schema.omg.org/spec/KDM/1.2/platform" name="Schema Example">
  <model xmi:id="id.0" xmi:type="data:DataModel" name="Contracts">
```

```
<dataElement xmi:id="id.1" xmi:type="data:RelationalSchema" name="Contracts">
  <dataElement xmi:id="id.2" xmi:type="data:RelationalTable" name="products">
    <dataElement xmi:id="id.3" xmi:type="data:UniqueKey" name="ID" implementation="id.4"/>
    <itemUnit xmi:id="id.4" name="ID" type="id.57"/>
    <itemUnit xmi:id="id.5" name="name" type="id.58"/>
    <itemUnit xmi:id="id.6" name="type" type="id.58"/>
  </dataElement>
  <dataElement xmi:id="id.7" xmi:type="data:RelationalTable" name="contracts">
    <dataElement xmi:id="id.8" xmi:type="data:UniqueKey" name="ID" implementation="id.11"/>
    <dataElement xmi:id="id.9" xmi:type="data:ReferenceKey" implementation="id.12">
      <dataRelation xmi:id="id.10" xmi:type="data:KeyRelation" to="id.3" from="id.9"/>
    </dataElement>
    <itemUnit xmi:id="id.11" name="ID" type="id.57"/>
    <itemUnit xmi:id="id.12" name="product" type="id.57"/>
    <itemUnit xmi:id="id.13" name="revenue" type="id.59"/>
    <itemUnit xmi:id="id.14" name="dateSigned" type="id.60"/>
  </dataElement>
  <dataElement xmi:id="id.15" xmi:type="data:RelationalTable" name="revenueRecognitions">
    <dataElement xmi:id="id.16" xmi:type="data:UniqueKey" implementation="id.25 id.27"/>
    <dataElement xmi:id="id.17" xmi:type="data:ReferenceKey" implementation="id.25">
      <dataRelation xmi:id="id.18" xmi:type="data:KeyRelation" to="id.8" from="id.17"/>
    </dataElement>
    <dataElement xmi:id="id.19" xmi:type="data:DataEvent" name="e1" kind="Insert">
      <abstraction xmi:id="id.20" name="e1.1" kind="Call">
        <actionRelation xmi:id="id.21" xmi:type="action:Calls" to="id.47" from="id.20"/>
      </abstraction>
    </dataElement>
    <dataElement xmi:id="id.22" xmi:type="data:DataEvent" name="e2" kind="Update">
      <abstraction xmi:id="id.23" name="e2.1" kind="Call">
        <actionRelation xmi:id="id.24" xmi:type="action:Calls" to="id.47" from="id.23"/>
      </abstraction>
    </dataElement>
    <itemUnit xmi:id="id.25" name="contract" type="id.57"/>
    <itemUnit xmi:id="id.26" name="amount" type="id.59"/>
    <itemUnit xmi:id="id.27" name="recognizedOn" type="id.60"/>
  </dataElement>
  <codeElement xmi:id="id.28" xmi:type="code:CallableUnit" name="INSERT_RECOGNITIONS" kind="regular">
    <entryFlow xmi:id="id.29" to="id.35" from="id.28"/>
    <codeElement xmi:id="id.30" xmi:type="code:Signature">
      <parameterUnit xmi:id="id.31" name="contractID" type="id.57" pos="1"/>
      <parameterUnit xmi:id="id.32" name="amount" type="id.59" pos="2"/>
      <parameterUnit xmi:id="id.33" name="recognizedOn" type="id.60" pos="3"/>
      <parameterUnit xmi:id="id.34" name="result" type="id.57" kind="byReference" pos="4"/>
    </codeElement>
    <codeElement xmi:id="id.35" xmi:type="action:ActionElement" name="a1" kind="Insert">
      <source xmi:id="id.36" language="SQL"
        snippet="INSERT INTO revenueRecognitions VALUES( contractID, amount, recognizedOn);"/>
      <actionRelation xmi:id="id.37" xmi:type="action:Reads" to="id.31" from="id.35"/>
      <actionRelation xmi:id="id.38" xmi:type="action:Reads" to="id.32" from="id.35"/>
      <actionRelation xmi:id="id.39" xmi:type="action:Reads" to="id.33" from="id.35"/>
      <actionRelation xmi:id="id.40" xmi:type="data:WritesColumnSet" to="id.15" from="id.35"/>
      <actionRelation xmi:id="id.41" xmi:type="data:ProducesDataEvent" to="id.19" from="id.35"/>
    </codeElement>
    <codeElement xmi:id="id.42" xmi:type="action:ActionElement" name="a2" kind="Assign">
      <source xmi:id="id.43" language="SQL" snippet="SET result = 1;"/>
```

```
        <codeElement xmi:id="id.44" xmi:type="code:Value" name="1"/>
        <actionRelation xmi:id="id.45" xmi:type="action:Reads" to="id.44" from="id.42"/>
        <actionRelation xmi:id="id.46" xmi:type="action:Writes" to="id.34" from="id.42"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.47" xmi:type="code:CallableUnit" name="reminder1">
      <entryFlow xmi:id="id.48" to="id.49" from="id.47"/>
      <codeElement xmi:id="id.49" xmi:type="action:ActionElement" name="a3" kind="Throw">
        <codeElement xmi:id="id.50" xmi:type="code:ValueList" name="error">
          <valueElement xmi:id="id.51" xmi:type="code:Value"
               name="&amp;quot;Notify sales!&amp;quot;" type="id.58"/>
          <valueElement xmi:id="id.52" xmi:type="code:Value" name="16" type="id.57"/>
          <valueElement xmi:id="id.53" xmi:type="code:Value" name="10" type="id.57"/>
        </codeElement>
        <actionRelation xmi:id="id.54" xmi:type="action:Throws" to="id.50" from="id.49"/>
      </codeElement>
    </codeElement>
  </dataElement>
 </model>
 <model xmi:id="id.55" xmi:type="code:CodeModel">
  <codeElement xmi:id="id.56" xmi:type="code:LanguageUnit" name="SQL datatypes">
    <codeElement xmi:id="id.57" xmi:type="code:IntegerType" name="sql int"/>
    <codeElement xmi:id="id.58" xmi:type="code:StringType" name="sql varchar"/>
    <codeElement xmi:id="id.59" xmi:type="code:DecimalType" name="sql decimal"/>
    <codeElement xmi:id="id.60" xmi:type="code:DateType" name="sql date"/>
    <codeElement xmi:id="id.61" xmi:type="code:BooleanType"/>
  </codeElement>
 </model>
 <model xmi:id="id.62" xmi:type="platform:PlatformModel">
  <platformElement xmi:id="id.63" xmi:type="platform:ExternalActor">
    <abstraction xmi:id="id.64" >
      <actionRelation xmi:id="id.65" xmi:type="data:ProducesDataEvent" to="id.19" from="id.64"/>
    </abstraction>
  </platformElement>
 </model>
</kdm:Segment>
```

18.6.3 RelationalView Class

A RelationalView class is a specific subclass of the ColumnSet class that represents Views of relational databases. A view is a virtual table whose contents are defined by a query. Like a real table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases. Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.

Superclass

ColumnSet

Semantics

A view can be thought of as either a virtual table or a stored query. Unless a view is indexed, its data is not stored in the database as a distinct object. What is stored in the database is a SELECT statement. The result set of the SELECT statement forms the virtual table returned by the view. A user can use this virtual table by referencing the view name in SQL statements the same way a table is referenced. Usually there are no restrictions on querying through views and few restrictions on modifying data through them.

In KDM, a RelationalView owns ItemUnits that correspond to the fields of the virtual table. An “abstracted” action of the View can store the corresponding SELECT statement.

18.6.4 DataSegment Class

A DataSegment class is a meta-model element that represents a segment of a hierarchical database, such as IMS.

Superclass

ColumnSet

Semantics

A hierarchical database is a kind of database management system that links records together in a tree data structure such that each record type has only one owner. Hierarchical structures were widely used in the first mainframe database management systems. However, due to their restrictions, they often cannot be used to relate structures that exist in the real world.

A database segment defines the fields for a set of segment instances similar to the way a relational table defines columns for a set of rows in a table. In this way, segments relate to relational tables, and fields in a segment relate to columns in a relational table.

Example (IMS):

```
DLR_PCB1 PCB TYPE=DB,DBDNAME=DEALERDB,PROCOPT=GO,KEYLEN=42
SENSEG NAME=DEALER,PARENT=0
SENSEG NAME=MODEL,PARENT=DEALER
SENSEG NAME=ORDER,PARENT=MODEL
SENSEG NAME=SALES,PARENT=MODEL
SENSEG NAME=STOCK,PARENT=MODEL
PSBGEN PSBNAME=DLR_PSB,MAXQ=200,LANG=JAVA
END

DBD NAME=DEALERDB,ACCESS=(HDAM,OSAM),RMNAME=(DFSHDC40.1.10)
SEGM NAME=DEALER,PARENT=0,BYTES=94,
FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
FIELD NAME=DLRNAME,BYTES=30,START=5,TYPE=C
SEGM NAME=MODEL,PARENT=DEALER,BYTES=43
FIELD NAME=(MODTYPE,SEQ,U),BYTES=2,START=1,TYPE=C
FIELD NAME=MAKE,BYTES=10,START=3,TYPE=C
FIELD NAME=MODEL,BYTES=10,START=13,TYPE=C
FIELD NAME=YEAR,BYTES=4,START=23,TYPE=C
FIELD NAME=MSRP,BYTES=5,START=27,TYPE=P
SEGM NAME=ORDER,PARENT=MODEL,BYTES=127
FIELD NAME=(ORDNBR,SEQ,U),BYTES=6,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=50,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=75,TYPE=C
```

```
SEGM NAME=SALES,PARENT=MODEL,BYTES=113
FIELD NAME=(SALDATE,SEQ,U),BYTES=8,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=9,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=34,TYPE=C
FIELD NAME=STKVIN,BYTES=20,START=94,TYPE=C
SEGM NAME=STOCK,PARENT=MODEL,BYTES=62
FIELD NAME=(STKVIN,SEQ,U),BYTES=20,START=1,TYPE=C
FIELD NAME=COLOR,BYTES=10,START=37,TYPE=C
FIELD NAME=PRICE,BYTES=5,START=47,TYPE=C
FIELD NAME=LOT,BYTES=10,START=52,TYPE=C
DBDGEN
FINISH
END
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="IMS Example">
  <model xmi:id="id.0" xmi:type="data:DataModel">
    <dataElement xmi:id="id.1" xmi:type="data:Catalog" name="DEALERDB">
      <dataElement xmi:id="id.2" xmi:type="data:DataSegment" name="Dealer">
        <dataElement xmi:id="id.3" xmi:type="data:DataSegment" name="Model">
          <dataElement xmi:id="id.4" xmi:type="data:DataSegment" name="Order">
            <dataElement xmi:id="id.5" xmi:type="data:UniqueKey" implementation="id.6"/>
            <itemUnit xmi:id="id.6" name="ORDNBR" type="id.30" size="2"/>
            <itemUnit xmi:id="id.7" name="LASTNME" type="id.30" size="25"/>
            <itemUnit xmi:id="id.8" name="FIRSTNME" type="id.30" size="25"/>
          </dataElement>
          <dataElement xmi:id="id.9" xmi:type="data:DataSegment" name="Sales">
            <dataElement xmi:id="id.10" xmi:type="data:UniqueKey" implementation="id.11"/>
            <itemUnit xmi:id="id.11" name="SALDATE" type="id.30" size="8"/>
            <itemUnit xmi:id="id.12" name="LASTNME" type="id.30" size="25"/>
            <itemUnit xmi:id="id.13" name="FIRSTNME" type="id.30" size="25"/>
            <itemUnit xmi:id="id.14" name="STKVIN" type="id.30" size="20"/>
          </dataElement>
          <dataElement xmi:id="id.15" xmi:type="data:DataSegment" name="Stock">
            <dataElement xmi:id="id.16" xmi:type="data:UniqueKey" implementation="id.17"/>
            <itemUnit xmi:id="id.17" name="STKVIN" type="id.30" size="20"/>
            <itemUnit xmi:id="id.18" name="COLOR" type="id.30" size="10"/>
            <itemUnit xmi:id="id.19" name="PRICE" type="id.30" size="5"/>
            <itemUnit xmi:id="id.20" name="LOT" type="id.30" size="10"/>
          </dataElement>
          <dataElement xmi:id="id.21" xmi:type="data:UniqueKey" implementation="id.22"/>
          <itemUnit xmi:id="id.22" name="MODTYPE" type="id.30" size="2"/>
          <itemUnit xmi:id="id.23" name="MAKE" size="10"/>
          <itemUnit xmi:id="id.24" name="YEAR" size="4"/>
          <itemUnit xmi:id="id.25" name="MSRP" type="id.31" size="5"/>
        </dataElement>
        <dataElement xmi:id="id.26" xmi:type="data:UniqueKey" implementation="id.27"/>
        <itemUnit xmi:id="id.27" name="DRLNO" type="id.30" size="4"/>
        <itemUnit xmi:id="id.28" name="DLRNAME" size="30"/>
      </dataElement>
    </dataElement>
```

```
  </model>
  <model xmi:id="id.29" xmi:type="code:CodeModel" name="Common IMS datatypes">
    <codeElement xmi:id="id.30" xmi:type="code:StringType" name="IMS type c"/>
    <codeElement xmi:id="id.31" xmi:type="code:DecimalType" name="IMS type packeddecimal"/>
  </model>
</kdm:Segment>
```

18.6.5 RecordFile Class

The RecordFile class is a meta-model element that represents files as a set of records. RecordFile can be indexed or sequential.

Superclass

ColumnSet

Semantics

In a non-relational database system, a record is an entry in a file, consisting of individual elements of information, which together provide full details about an aspect of the information needed by the system. Individual elements are held in fields and all records are held in files. An example of a record might be an employee. Every detail of the employee, for example, date of birth, department code, or full names will be found in a number of fields. A file is a set of records, where each record is a sequence of fields. A sequential file is a computer file storage format in which one record follows another. Records can be accessed sequentially only. It is required with magnetic tape. An indexed file owns one or more indexes that allow records to be retrieved by a specific value or in a particular sort order.

Example (cobol)

The following example illustrates the representation of RecordFile. The CodeModel of this example is incomplete as it focuses on the DataModel, and well as combined representation involving the CodeModel, DataModel, PlatformModel, and EventModel.

```
FILE-CONTROL.
   SELECT SEQUENTIAL-FILE ASSIGN TO 'A:\SEQ.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
   SELECT INDEXED-FILE
      ASSIGN TO 'A:\INDMAST.DAT'
      ORGANIZATION IS INDEXED
      ASSESS IS SEQUENTIAL
      RECORD KEY IS IND-SOC-SEC-NUM
      FILE STATUS IS INDEXED-STATUS-BYTES.

FILE SECTION.
FD SEQUENTIAL FILE
   RECORD COTNAINS 39 CHARACTERS
   DATA RECORD IS SEQUENTIAL-RECORD.
01 SEQUENTIAL-RECORD.
        05 SEQ-SOC-SEC-NUM PIC X(9).
        05 SEQ-REST-OF-RECORDPIC X(30).

FD INDEXED-FILE
        RECORD CONTAINS 39 CHARACTERS
        DATA RECORD IS INDEXED-RECORD.
01  INDEXED-RECORD.
```

```
        05 IND-SOC-SEC-NUM  PIC X(9).
        05 IND-REST-OF-RECORDPIC X(30).

PROCEDURE DIVISION.
0010-UPDATE-MASTER-FILE.
        OPEN INPUT SEQUENTIAL-FILE
            OUTPUT INDEXED-FILE.
        PERFORM UNTIL END-OF-FILE-SWITCH = 'YES'
            READ SEQUENTIAL-FILE
                AT END
                    MOVE 'YES' TO END-OF-FILE-SWITCH
                NOT AT END
                    MOVE SEQ-SOC-SEC-NUM TO IND-SOC-SEC-NUM
                    MOVE SEQ-REST-OF-RECORD TO IND-REST-OF-RECORD
                    WRITE INDEXED-RECORD
                      INVALID KEY PERFORM 0020-EXPLAIN-WRITE-ERROR
                    END-WRITE
            END-READ
        END-PERFORM.
        CLOSE SEQUENTIAL-FILE
                INDEXED-FILE.
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
        xmlns:event="http://schema.omg.org/spec/KDM/1.2/event"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
        xmlns:platform="http://schema.omg.org/spec/KDM/1.2/platform" name="RecordFile example">
  <model xmi:id="id.0" xmi:type="data:DataModel">
    <dataElement xmi:id="id.1" xmi:type="data:RecordFile" name="SEQUENTIAL-FILE">
      <itemUnit xmi:id="id.2" name="SEQ-SOC-SEC-NUM" type="id.115" ext="PIC X(9)" size="9"/>
      <itemUnit xmi:id="id.3" name="SEQ-REST-OF-RECORD" type="id.115" ext="PIC X(30)" size="30"/>
    </dataElement>
    <dataElement xmi:id="id.4" xmi:type="data:RecordFile" name="INDEXED-FILE">
      <dataElement xmi:id="id.5" xmi:type="data:UniqueKey" implementation="id.7"/>
      <dataElement xmi:id="id.6" xmi:type="data:Index" implementation="id.7"/>
      <itemUnit xmi:id="id.7" name="IND-SOC-SEC-NUM" type="id.115" ext="PIC X(9)" size="9"/>
      <itemUnit xmi:id="id.8" name="IND-REST-OF-RECORD" type="id.115" ext="PIC X(30)" size="30"/>
    </dataElement>
    <dataElement xmi:id="id.9" xmi:type="data:DataAction" name="da1" kind="open" implementation="id.44">
      <abstraction xmi:id="id.10" name="da1" kind="open">
        <actionRelation xmi:id="id.11" xmi:type="data:ManagesData" to="id.1" from="id.10"/>
        <actionRelation xmi:id="id.12" xmi:type="platform:ManagesResource" to="id.75" from="id.10"/>
      </abstraction>
    </dataElement>
    <dataElement xmi:id="id.13" xmi:type="data:DataAction" name="da2" kind="open" implementation="id.44">
      <abstraction xmi:id="id.14" name="da2" kind="open">
        <actionRelation xmi:id="id.15" xmi:type="platform:ManagesResource" to="id.79" from="id.14"/>
        <actionRelation xmi:id="id.16" xmi:type="data:ManagesData" to="id.4" from="id.14"/>
      </abstraction>
    </dataElement>
    <dataElement xmi:id="id.17" xmi:type="data:DataAction" name="da3" kind="read" implementation="id.47">
```

```
    <abstraction xmi:id="id.18" name="da3" kind="read">
      <actionRelation xmi:id="id.19" xmi:type="data:ReadsColumnSet" to="id.1" from="id.18"/>
      <actionRelation xmi:id="id.20" xmi:type="action:Writes" to="id.2" from="id.18"/>
      <actionRelation xmi:id="id.21" xmi:type="action:Writes" to="id.3" from="id.18"/>
      <actionRelation xmi:id="id.22" xmi:type="platform:ReadsResource" to="id.75" from="id.18"/>
    </abstraction>
    <dataElement xmi:id="id.23" name="at end" kind="EOF">
      <abstraction xmi:id="id.24" name="ae1">
        <actionRelation xmi:id="id.25" xmi:type="action:ExceptionFlow" to="id.50" from="id.24"/>
      </abstraction>
    </dataElement>
    <dataElement xmi:id="id.26" name="not at end" kind="NOT EOF">
      <abstraction xmi:id="id.27" name="nae1">
        <actionRelation xmi:id="id.28" xmi:type="action:Flow" to="id.53" from="id.27"/>
      </abstraction>
    </dataElement>
  </dataElement>
  <dataElement xmi:id="id.29" xmi:type="data:DataAction" name="da4" kind="write"
               implementation="id.59">
    <abstraction xmi:id="id.30" name="da4" kind="write">
      <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.7" from="id.30"/>
      <actionRelation xmi:id="id.32" xmi:type="action:Reads" to="id.8" from="id.30"/>
      <actionRelation xmi:id="id.33" xmi:type="data:WritesColumnSet" to="id.4" from="id.30"/>
      <actionRelation xmi:id="id.34" xmi:type="platform:WritesResource" to="id.79" from="id.30"/>
    </abstraction>
    <dataElement xmi:id="id.35" name="invalid key" kind="INVALID KEY">
      <abstraction xmi:id="id.36" name="ik1">
        <actionRelation xmi:id="id.37" xmi:type="action:ExceptionFlow" to="id.62" from="id.36"/>
      </abstraction>
    </dataElement>
  </dataElement>
  <dataElement xmi:id="id.38" xmi:type="data:DataAction" name="da5" kind="close">
    <abstraction xmi:id="id.39" name="da5" kind="close"/>
  </dataElement>
  <dataElement xmi:id="id.40" xmi:type="data:DataAction" name="da6" kind="close">
    <abstraction xmi:id="id.41" name="da5" kind="close"/>
  </dataElement>
</model>
<model xmi:id="id.42" xmi:type="code:CodeModel">
  <codeElement xmi:id="id.43" xmi:type="code:CodeAssembly">
    <codeElement xmi:id="id.44" xmi:type="action:ActionElement" name="a0" kind="open">
      <source xmi:id="id.45" language="Cobol"
          snippet="OPEN INPUT SEQUENTIAL-FILE OUTPUT INDEXED-FILE."/>
      <actionRelation xmi:id="id.46" xmi:type="action:Flow" to="id.47" from="id.44"/>
    </codeElement>
    <codeElement xmi:id="id.47" xmi:type="action:ActionElement" name="a1" kind="read">
      <source xmi:id="id.48" language="Cobol" snippet="READ SEQUENTIAL-FILE"/>
      <actionRelation xmi:id="id.49" xmi:type="action:Flow" to="id.53" from="id.47"/>
    </codeElement>
    <codeElement xmi:id="id.50" xmi:type="action:ActionElement" name="a2">
      <source xmi:id="id.51" language="Cobol" snippet="MOVE 'YES' TO END-OF-FILE-SWITCH"/>
      <actionRelation xmi:id="id.52" xmi:type="action:Flow" to="id.64" from="id.50"/>
    </codeElement>
    <codeElement xmi:id="id.53" xmi:type="action:ActionElement" name="a3">
      <source xmi:id="id.54" language="Cobol" snippet="MOVE SEQ-SOC-SEC-NUM TO IND-SOC-SEQ-NUM"/>
```

```
      <actionRelation xmi:id="id.55" xmi:type="action:Flow" to="id.56" from="id.53"/>
    </codeElement>
    <codeElement xmi:id="id.56" xmi:type="action:ActionElement" name="a4">
      <source xmi:id="id.57" language="Cobol" snippet="MOVE SEQ-REST-OF-RECORD TO IND-REST-OF-RECORD"/>
      <actionRelation xmi:id="id.58" xmi:type="action:Flow" to="id.59" from="id.56"/>
    </codeElement>
    <codeElement xmi:id="id.59" xmi:type="action:ActionElement" name="a5" kind="call">
      <source xmi:id="id.60" language="Cobol" snippet="WRITE INDEXED-RECORD"/>
      <actionRelation xmi:id="id.61" xmi:type="action:Flow" to="id.64" from="id.59"/>
    </codeElement>
    <codeElement xmi:id="id.62" xmi:type="action:ActionElement" name="a6" kind="write">
      <source xmi:id="id.63" language="Cobol" snippet="PERFORM 0020-EXPLAIN-WRITE-ERROR"/>
    </codeElement>
    <codeElement xmi:id="id.64" xmi:type="action:ActionElement" name="a7" kind="write">
      <source xmi:id="id.65" language="Cobol" snippet="UNTIL END-OF-FILE-SWITCH = 'YES'"/>
      <actionRelation xmi:id="id.66" xmi:type="action:FalseFlow" to="id.47" from="id.64"/>
      <actionRelation xmi:id="id.67" xmi:type="action:TrueFlow" to="id.68" from="id.64"/>
    </codeElement>
    <codeElement xmi:id="id.68" xmi:type="action:ActionElement" name="a8" kind="close">
      <source xmi:id="id.69" language="Cobol" snippet="Close SEQUENTIAL-FILE INDEXED-FILE."/>
    </codeElement>
  </codeElement>
</model>
<model xmi:id="id.70" xmi:type="platform:PlatformModel">
  <platformElement xmi:id="id.71" xmi:type="platform:DeployedSoftwareSystem" groupedComponent="id.73"/>
  <platformElement xmi:id="id.72" xmi:type="platform:Machine">
    <deployedComponent xmi:id="id.73" groupedCode="id.43"/>
    <deployedResource xmi:id="id.74" >
      <platformElement xmi:id="id.75" xmi:type="platform:StreamResource">
        <abstraction xmi:id="id.76" name="ra1" kind="">
          <actionRelation xmi:id="id.77" xmi:type="data:HasContent" to="id.1" from="id.76"/>
          <actionRelation xmi:id="id.78" xmi:type="event:HasState" to="id.89" from="id.76"/>
        </abstraction>
      </platformElement>
      <platformElement xmi:id="id.79" xmi:type="platform:FileResource">
        <abstraction xmi:id="id.80" name="ra2" kind="">
          <actionRelation xmi:id="id.81" xmi:type="data:HasContent" to="id.4" from="id.80"/>
        </abstraction>
      </platformElement>
    </deployedResource>
  </platformElement>
  <platformElement xmi:id="id.82" xmi:type="platform:PlatformAction" name="pa1" kind="open">
    <abstraction xmi:id="id.83" name="pa1">
      <actionRelation xmi:id="id.84" xmi:type="platform:ManagesResource" to="id.75" from="id.83"/>
    </abstraction>
  </platformElement>
  <platformElement xmi:id="id.85" xmi:type="platform:PlatformAction" name="pa2" kind="open">
    <abstraction xmi:id="id.86" name="pa2">
      <actionRelation xmi:id="id.87" xmi:type="platform:ManagesResource" to="id.79" from="id.86"/>
    </abstraction>
  </platformElement>
</model>
<model xmi:id="id.88" xmi:type="event:EventModel">
  <eventElement xmi:id="id.89" xmi:type="event:EventResource" name="sequential-file">
    <eventElement xmi:id="id.90" xmi:type="event:State" name="closed">
```

```
        <eventElement xmi:id="id.91" xmi:type="event:Transition" name="tr1">
          <eventRelation xmi:id="id.92" xmi:type="event:ConsumesEvent" to="id.110" from="id.91"/>
          <eventRelation xmi:id="id.93" xmi:type="event:NextState" to="id.103" from="id.91"/>
          <eventRelation xmi:id="id.94" xmi:type="event:NextState" to="id.95" from="id.91"/>
        </eventElement>
      </eventElement>
      <eventElement xmi:id="id.95" xmi:type="event:State" name="opened.not at end">
        <eventElement xmi:id="id.96" xmi:type="event:Transition" name="tr2">
          <eventRelation xmi:id="id.97" xmi:type="event:ConsumesEvent" to="id.111" from="id.96"/>
          <eventRelation xmi:id="id.98" xmi:type="event:NextState" to="id.103" from="id.96"/>
          <eventRelation xmi:id="id.99" xmi:type="event:NextState" to="id.95" from="id.96"/>
        </eventElement>
        <eventElement xmi:id="id.100" xmi:type="event:Transition" name="tr3">
          <eventRelation xmi:id="id.101" xmi:type="event:ConsumesEvent" to="id.112" from="id.100"/>
          <eventRelation xmi:id="id.102" xmi:type="event:NextState" to="id.90" from="id.100"/>
        </eventElement>
      </eventElement>
      <eventElement xmi:id="id.103" xmi:type="event:State" name="opened.at end">
        <eventElement xmi:id="id.104" xmi:type="event:Transition" name="tr4">
          <eventRelation xmi:id="id.105" xmi:type="event:ConsumesEvent" to="id.112" from="id.104"/>
          <eventRelation xmi:id="id.106" xmi:type="event:NextState" to="id.90" from="id.104"/>
        </eventElement>
        <eventElement xmi:id="id.107" xmi:type="event:Transition" name="tr5">
          <eventRelation xmi:id="id.108" xmi:type="event:ConsumesEvent" to="id.111" from="id.107"/>
          <eventRelation xmi:id="id.109" xmi:type="event:NextState" to="id.103" from="id.107"/>
        </eventElement>
      </eventElement>
      <eventElement xmi:id="id.110" xmi:type="event:Event" name="open" kind="open"/>
      <eventElement xmi:id="id.111" xmi:type="event:Event" name="read"/>
      <eventElement xmi:id="id.112" xmi:type="event:Event" name="close"/>
    </eventElement>
  </model>
  <model xmi:id="id.113" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.114" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.115" xmi:type="code:StringType" name="X"/>
    </codeElement>
  </model>
</kdm:Segment>
```

18.7 KeyIndex Class Diagram

The KeyIndex class diagram collects together classes and associations of the Data package. They provide basic meta-model constructs to define the various data related relationships.

The class diagram shown in Figure 18.5 captures these classes and their relations.

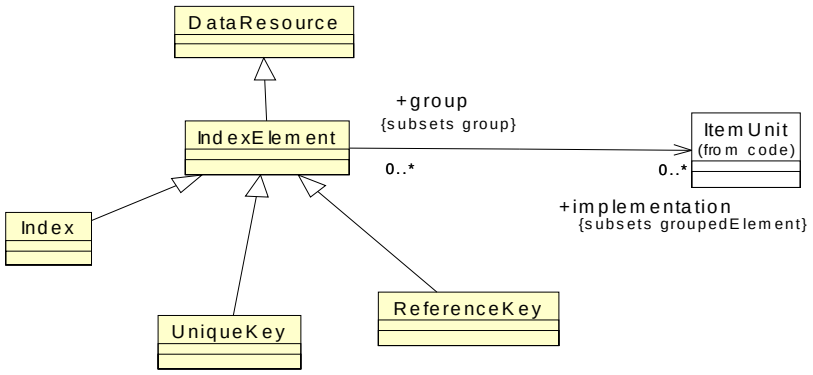

Figure 18.5 - KeyIndex Class Diagram

18.7.1 IndexElement Class (generic)

IndexElement class is a generic meta-model element that defines the common properties of the index items and key items of persistent data stores. IndexElement uses the KDM group mechanism. IndexElement is subclassed by concrete classes with more precise semantics. IndexElement is itself a concrete class that can be used as an extended meta-model element with an appropriate stereotype to represent situations that do not fit into the semantics of the subclasses of the IndexElement.

Superclass

DataResource

Associations

implementation : ItemUnit[1]	the set of ItemUnits that constitute the index

Constraints:

1. Index owned by a data element should group elements that are owned by that data element.
2. IndexElement should have a stereotype.

Semantics

IndexElement defines a group of data elements that can be used as an endpoint for various data relationships.

18.7.2 UniqueKey Class

A UniqueKey is a meta-model element that represents primary keys in relational database tables, segments of hierarchical databases, or indexed files. UniqueKey is a group of columns.

Superclass

IndexElement

Constraints

1. UniqueKey owned by a data element should group ItemUnit elements that are owned by that data element.

Semantics

A UniqueKey represents the primary key to a certain table or relational database or certain fields in an indexed file. A primary key is one or more columns whose values uniquely identify every row in a table or every record in an indexed file. Normally an index always exists on the primary key.

18.7.3 ReferenceKey Class

A ReferenceKey is a meta-model element that represents foreign key in databases or indexed files. ReferenceKey is a group of columns.

Superclass

IndexElement

Constraints

1. ReferenceKey owned by a data element should group ItemUnit elements that are owned by that data element.

Semantics

A foreign key is the primary key of one data structure that is placed into a related data structure to represent a relationship among those structures. Foreign keys resolve relationships, and support navigation among data structures. ReferenceKey is a group of one or more columns in a relational database table or segment of a hierarchical database or an indexed file that implements a many-to-one relationship that the table, segment, or file in question has with another table, segment, or file, or with itself.

18.7.4 Index Class

An Index class is a meta-model element that represents an index to a relational or hierarchical database or an indexed file.

Superclass

IndexElement

Constraints

1. Index owned by a data element should group ItemUnit elements that are owned by that data element.

Semantics

Index is a mechanism to locate and access data within a database. An index may quote one or more columns and be a means of enforcing uniqueness on their values.

18.8 Key Relations Class Diagram

Figure 18.6 depicts the key relations within the Data Package. A Key is a way to access data without reading through an entire data structure sequentially.

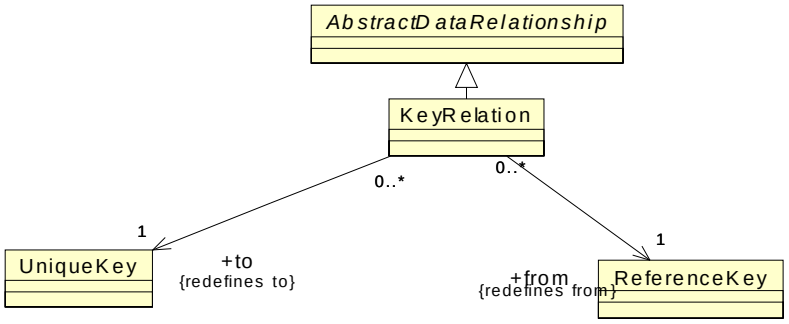

Figure 18.6 - KeyRelations Class Diagram

A KeyRelation class associates a ReferenceKey in one data container, with a UniqueKey in another container, which means that there is one and only one key value for that data.

18.8.1 KeyRelationship Class

A KeyRelationship is a meta-model element that represents an association between a ReferenceKey with the corresponding UniqueKey.

Superclass

AbstractDataRelationship

Associations

from : ReferenceKey[1]	Foreign key is a certain table, segment, or file.
to: UniqueKey[1]	Primary key is a certain table, segment, or key.

Semantics

ReferenceKey is a group of one or more columns in a relational database table or segment of a hierarchical database or an indexed file that implements a many-to-one relationship that the table, segment, or file in question has with another table, segment, or file, or with itself.

18.9 DataActions Class Diagram

DataAction class diagram defines a set of meta-model elements whose purpose is to represent semantic associations between the elements of data models, as well as associations between data models and other KDM models. Figure 18.7 depicts the key classes and association of the DataAction diagram. Data actions follow the common pattern of “resource actions.” Each data action is a “projection” of one or more action elements of the Code Model that uses some API to the runtime platform to manage data resources. Each data action is linked back to the corresponding action elements from one or more code models through the “implementation” association. Each data action may own one or more “abstracted” actions, which are used to model detailed resource related semantics.

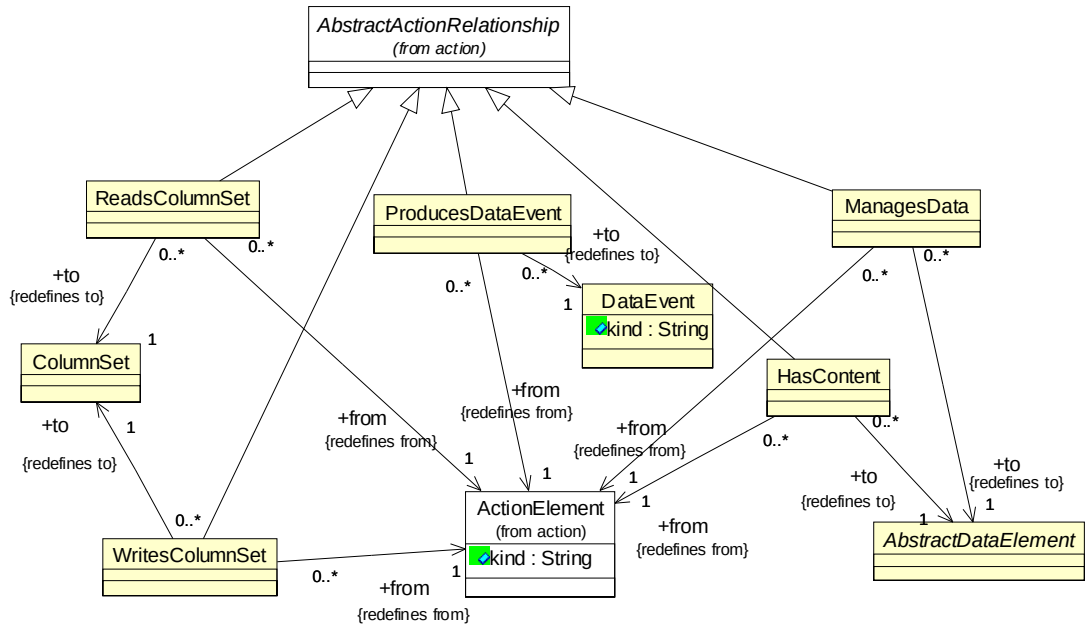

Figure 18.7 - DataActions Class Diagram

18.9.1 ReadsColumnSet Class

ReadsColumnSet class follows the pattern of a “resource action relationship.” It represents various types of accesses to data resources where there is a flow of data **from** the resource. ReadsColumnSet relationship is similar to Reads relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the DataAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:ColumnSet[1]	the data resource being accessed

Constraints

1. This relationship should not be used in Code models.

Semantics

ReadsColumnSet represents a data flow from a certain ColumnSet element to a data action.

18.9.2 WritesColumnSet Class

WritesColumnSet class follows the pattern of a “resource action relationship.” It represents various types of accesses to user interface resources where there is a flow of data to the resource. WritesColumnSet relationship is similar to Writes relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the DataAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:ColumnSet[1]	the data resource being accessed

Constraints

1. This relationship should not be used in Code models.

Semantics

WritesColumnSet represents a data flow from a data action to a certain ColumnSet element.

18.9.3 ManagesData Class

Manages class follows the pattern of a “resource action relationship.” It represents various types of accesses to user interface resources where there is no flow of data to or from the resource. ManagesData relationship is similar to Addresses relationship from Action Package. The nature of the operation on the resource is represented by the “kind” attribute of the DataAction that owns this relationship through the “abstracted” action container property.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:AbstractDataElement[1]	the data resource being accessed

Constraints

1. This relationship should not be used in Code models.

Semantics

Manages represents a certain change of state to a certain AbstractDataElement.

18.9.4 HasContent Class

HasContent class follows the pattern of a “resource action relationship.” HasContent is a structural relationship. It does not represent resource manipulations. HasContent relationship uses the “abstracted” action container mechanism to provide certain capabilities to other Resource Layer packages. “HasContent” relationship makes it possible to associate an element of a data model with any resource.

Superclass

Action::AbstractActionRelationship

Associations

from:ActionElement[1]	“abstracted” action owned by some resource
to:AbstractDataElement[1]	the data resource being accessed

Constraints

1. This relationship should not be used in Code models.

Semantics

HasContent represents an association between any KDM resource or behavior abstraction element (through the “abstracted” action mechanism) and the data element, describing the data organization related to this element.

Example (Java, embedded SQL, JDBC)

```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned date)

final String findContractStatement=
          "SELECT * FROM contracts c, products p" +
          "WHERE ID = ? AND c.product = p.ID ";

public void calculateRecognitions( long contractID ) {
         Connection db=DriverManager.getConnection("jdbc:odbc:foobar","sunny","");
         PreparedStatement stmt=db.prepareStatement(findContractStatement);
         stmt.setLong(1,contractID);
         ResultSet contracts=stmt.executeQuery();
         contracts.next();
         Money totalRevenue=Money.dollars(contracts.getBigDecimal("revenue"));
         MfDate recognitionDate=new MfDate(contracts.getDate("dateSigned"));
         }

<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
         xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
         xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
         xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
         xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
         xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
         xmlns:platform="http://schema.omg.org/spec/KDM/1.2/platform" name="Data Example">
  <model xmi:id="id.0" xmi:type="data:DataModel" name="Contracts">
    <dataElement xmi:id="id.1" xmi:type="data:RelationalSchema" name="Contracts">
```

```
  <dataElement xmi:id="id.2" xmi:type="data:RelationalTable" name="products">
    <dataElement xmi:id="id.3" xmi:type="data:UniqueKey" name="ID" implementation="id.4"/>
    <itemUnit xmi:id="id.4" name="ID" type="id.172"/>
    <itemUnit xmi:id="id.5" name="name" type="id.173"/>
    <itemUnit xmi:id="id.6" name="type" type="id.173"/>
  </dataElement>
  <dataElement xmi:id="id.7" xmi:type="data:RelationalTable" name="contracts">
    <dataElement xmi:id="id.8" xmi:type="data:UniqueKey" name="ID" implementation="id.11"/>
    <dataElement xmi:id="id.9" xmi:type="data:ReferenceKey" implementation="id.12">
      <dataRelation xmi:id="id.10" xmi:type="data:KeyRelation" to="id.3" from="id.9"/>
    </dataElement>
    <itemUnit xmi:id="id.11" name="ID" type="id.172"/>
    <itemUnit xmi:id="id.12" name="product" type="id.172"/>
    <itemUnit xmi:id="id.13" name="revenue" type="id.174"/>
    <itemUnit xmi:id="id.14" name="dateSigned" type="id.175"/>
  </dataElement>
</dataElement>
<dataElement xmi:id="id.15" xmi:type="data:DataAction" name="d1" kind="Connect"
         implementation="id.79">
  <abstraction xmi:id="id.16" name="da1" kind="Connect">
    <actionRelation xmi:id="id.17" xmi:type="action:Reads" to="id.80" from="id.16"/>
    <actionRelation xmi:id="id.18" xmi:type="action:Reads" to="id.81" from="id.16"/>
    <actionRelation xmi:id="id.19" xmi:type="action:Reads" to="id.82" from="id.16"/>
    <actionRelation xmi:id="id.20" xmi:type="platform:ManagesResource" to="id.67"/>
  </abstraction>
</dataElement>
<dataElement xmi:id="id.21" xmi:type="data:DataAction" name="d2" kind="Select"
        implementation="id.90 id.96 id.104">
  <source xmi:id="id.22" language="sql"
    snippet="&amp;quot;select * from contracts c, products p where ID = ? and c.product=p.ID  &amp;quot;"/>
  <abstraction xmi:id="id.23" name="w1" kind="Equal">
   <codeElement xmi:id="id.24" xmi:type="code:StorableUnit" name="t1" type="id.176" kind="register"/>
    <actionRelation xmi:id="id.25" xmi:type="action:Reads" to="id.11" from="id.23"/>
    <actionRelation xmi:id="id.26" xmi:type="action:Reads" to="id.77" from="id.23"/>
    <actionRelation xmi:id="id.27" xmi:type="action:Writes" to="id.24" from="id.23"/>
    <actionRelation xmi:id="id.28" xmi:type="action:Flow" to="id.29"/>
  </abstraction>
  <abstraction xmi:id="id.29" name="w2" kind="Equal">
   <codeElement xmi:id="id.30" xmi:type="code:StorableUnit" name="t2" type="id.176" kind="register"/>
    <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.12" from="id.29"/>
    <actionRelation xmi:id="id.32" xmi:type="action:Reads" to="id.4" from="id.29"/>
    <actionRelation xmi:id="id.33" xmi:type="action:Writes" from="id.29"/>
    <actionRelation xmi:id="id.34" xmi:type="action:Flow" to="id.35" from="id.29"/>
  </abstraction>
  <abstraction xmi:id="id.35" name="w3" kind="And">
   <codeElement xmi:id="id.36" xmi:type="code:StorableUnit" name="t3" type="id.176" kind="register"/>
    <actionRelation xmi:id="id.37" xmi:type="action:Reads" to="id.24" from="id.35"/>
    <actionRelation xmi:id="id.38" xmi:type="action:Reads" to="id.30"/>
    <actionRelation xmi:id="id.39" xmi:type="action:Flow" to="id.40" from="id.35"/>
  </abstraction>
  <abstraction xmi:id="id.40" name="w4" kind="Condition">
    <actionRelation xmi:id="id.41" xmi:type="action:TrueFlow" to="id.42" from="id.40"/>
  </abstraction>
  <abstraction xmi:id="id.42" name="s1" kind="Select">
    <actionRelation xmi:id="id.43" xmi:type="data:ReadsColumnSet" to="id.7" from="id.42"/>
```

```
      <actionRelation xmi:id="id.44" xmi:type="action:Reads" to="id.11" from="id.42"/>
      <actionRelation xmi:id="id.45" xmi:type="action:Reads" to="id.12" from="id.42"/>
      <actionRelation xmi:id="id.46" xmi:type="action:Reads" to="id.13" from="id.42"/>
      <actionRelation xmi:id="id.47" xmi:type="action:Reads" to="id.14" from="id.42"/>
      <actionRelation xmi:id="id.48" xmi:type="data:ReadsColumnSet" to="id.2"/>
      <actionRelation xmi:id="id.49" xmi:type="action:Reads" to="id.4" from="id.42"/>
      <actionRelation xmi:id="id.50" xmi:type="action:Reads" to="id.5" from="id.42"/>
      <actionRelation xmi:id="id.51" xmi:type="action:Reads" to="id.6" from="id.42"/>
      <actionRelation xmi:id="id.52" xmi:type="action:Writes" to="id.103" from="id.42"/>
      <actionRelation xmi:id="id.53" xmi:type="platform:ReadsResource" to="id.67" from="id.42"/>
    </abstraction>
  </dataElement>
  <dataElement xmi:id="id.54" xmi:type="data:DataAction" name="d3" kind="Retrieve"
       implementation="id.115">
    <abstraction xmi:id="id.55" name="da2" kind="Assign">
      <actionRelation xmi:id="id.56" xmi:type="action:Reads" to="id.13" from="id.55"/>
      <actionRelation xmi:id="id.57" xmi:type="action:Addresses" to="id.103" from="id.55"/>
      <actionRelation xmi:id="id.58" xmi:type="action:Writes" to="id.117" from="id.55"/>
    </abstraction>
  </dataElement>
  <dataElement xmi:id="id.59" xmi:type="data:DataAction" name="d4" kind="Retrieve"
       implementation="id.130">
    <abstraction xmi:id="id.60" name="da3" kind="Assign">
      <actionRelation xmi:id="id.61" xmi:type="action:Reads" to="id.14" from="id.60"/>
      <actionRelation xmi:id="id.62" xmi:type="action:Addresses" to="id.103" from="id.60"/>
      <actionRelation xmi:id="id.63" xmi:type="action:Writes" to="id.132" from="id.60"/>
    </abstraction>
  </dataElement>
</model>
<model xmi:id="id.64" xmi:type="platform:PlatformModel">
  <platformElement xmi:id="id.65" xmi:type="platform:Machine">
    <resource xmi:id="id.66" >
      <resource xmi:id="id.67" xmi:type="platform:DataManager" name="foobar">
        <abstraction xmi:id="id.68" name="dm1">
          <actionRelation xmi:id="id.69" xmi:type="data:HasContent" to="id.1"/>
        </abstraction>
      </resource>
    </resource>
  </platformElement>
</model>
<model xmi:id="id.70" xmi:type="code:CodeModel" name="Application">
  <codeElement xmi:id="id.71" xmi:type="code:ClassUnit" name="DataExample">
    <codeElement xmi:id="id.72" xmi:type="code:MemberUnit" name="findContractStatement">
      <codeRelation xmi:id="id.73" xmi:type="code:HasValue" to="id.145" from="id.72"/>
    </codeElement>
    <codeElement xmi:id="id.74" xmi:type="code:MethodUnit" name="calculateRecognitions">
      <entryFlow xmi:id="id.75" to="id.79" from="id.74"/>
      <codeElement xmi:id="id.76" xmi:type="code:Signature">
        <parameterUnit xmi:id="id.77" name="contractNumber" type="id.179"/>
      </codeElement>
      <codeElement xmi:id="id.78" xmi:type="code:StorableUnit" name="db" type="id.155" kind="local"/>
      <codeElement xmi:id="id.79" xmi:type="action:ActionElement" name="c1" kind="Call">
        <codeElement xmi:id="id.80" xmi:type="code:Value" name="&amp;quot;jdbc:odbc:foobar&amp;quot;"/>
        <codeElement xmi:id="id.81" xmi:type="code:Value" name="&amp;quot;sunny&amp;quot;" type="id.178"/>
        <codeElement xmi:id="id.82" xmi:type="code:Value" name="&amp;quot;&amp;quot;" type="id.178"/>
```

```
  <actionRelation xmi:id="id.83" xmi:type="action:Reads" to="id.80" from="id.79"/>
  <actionRelation xmi:id="id.84" xmi:type="action:Reads" to="id.81" from="id.79"/>
  <actionRelation xmi:id="id.85" xmi:type="action:Reads" to="id.82" from="id.79"/>
  <actionRelation xmi:id="id.86" xmi:type="action:Calls" to="id.154" from="id.79"/>
  <actionRelation xmi:id="id.87" xmi:type="action:Writes" to="id.78" from="id.79"/>
  <actionRelation xmi:id="id.88" xmi:type="action:Flow" to="id.90" from="id.79"/>
</codeElement>
<codeElement xmi:id="id.89" xmi:type="code:StorableUnit" name="stmt" type="id.161" kind="local"/>
<codeElement xmi:id="id.90" xmi:type="action:ActionElement" name="c2" kind="MethodCall">
  <actionRelation xmi:id="id.91" xmi:type="action:Addresses" to="id.78" from="id.90"/>
  <actionRelation xmi:id="id.92" xmi:type="action:Reads" to="id.72" from="id.90"/>
  <actionRelation xmi:id="id.93" xmi:type="action:Calls" to="id.156" from="id.90"/>
  <actionRelation xmi:id="id.94" xmi:type="action:Writes" to="id.89" from="id.90"/>
  <actionRelation xmi:id="id.95" xmi:type="action:Flow" to="id.96" from="id.90"/>
</codeElement>
<codeElement xmi:id="id.96" xmi:type="action:ActionElement" name="c3" kind="MethodCall">
  <codeElement xmi:id="id.97" xmi:type="code:Value" name="1"/>
  <actionRelation xmi:id="id.98" xmi:type="action:Addresses" to="id.89" from="id.96"/>
  <actionRelation xmi:id="id.99" xmi:type="action:Reads" to="id.97" from="id.96"/>
  <actionRelation xmi:id="id.100" xmi:type="action:Reads" to="id.77" from="id.96"/>
  <actionRelation xmi:id="id.101" xmi:type="action:Calls" to="id.162" from="id.96"/>
  <actionRelation xmi:id="id.102" xmi:type="action:Flow" to="id.104" from="id.96"/>
</codeElement>
<codeElement xmi:id="id.103" xmi:type="code:StorableUnit" name="contracts" type="id.157"
             kind="local"/>
<codeElement xmi:id="id.104" xmi:type="action:ActionElement" name="c4" kind="MethodCall">
  <actionRelation xmi:id="id.105" xmi:type="action:Addresses" to="id.89" from="id.104"/>
  <actionRelation xmi:id="id.106" xmi:type="action:Calls" to="id.163" from="id.104"/>
  <actionRelation xmi:id="id.107" xmi:type="action:Writes" to="id.103" from="id.104"/>
  <actionRelation xmi:id="id.108" xmi:type="action:Flow" to="id.109" from="id.104"/>
</codeElement>
<codeElement xmi:id="id.109" xmi:type="action:ActionElement" name="c5" kind="MethodCall">
  <actionRelation xmi:id="id.110" xmi:type="action:Addresses" to="id.103" from="id.109"/>
  <actionRelation xmi:id="id.111" xmi:type="action:Calls" to="id.158" from="id.109"/>
  <actionRelation xmi:id="id.112" xmi:type="action:Flow" to="id.114" from="id.109"/>
</codeElement>
<codeElement xmi:id="id.113" xmi:type="code:StorableUnit" name="totalRevenue" type="id.165"
             kind="local"/>
<codeElement xmi:id="id.114" xmi:type="action:ActionElement" name="c6" kind="Compound">
  <codeElement xmi:id="id.115" xmi:type="action:ActionElement" name="c6.1" kind="Call">
    <codeElement xmi:id="id.116" xmi:type="code:Value" name="&quot;revenue&quot;"/>
    <codeElement xmi:id="id.117" xmi:type="code:StorableUnit" name="t4" kind="register"/>
    <actionRelation xmi:id="id.118" xmi:type="action:Addresses" to="id.103" from="id.115"/>
    <actionRelation xmi:id="id.119" xmi:type="action:Calls" to="id.159" from="id.115"/>
    <actionRelation xmi:id="id.120" xmi:type="action:Writes" to="id.117" from="id.115"/>
    <actionRelation xmi:id="id.121" xmi:type="action:Flow" to="id.122" from="id.115"/>
  </codeElement>
  <codeElement xmi:id="id.122" xmi:type="action:ActionElement" name="c6.2" kind="Call">
    <actionRelation xmi:id="id.123" xmi:type="action:Reads" to="id.117" from="id.122"/>
    <actionRelation xmi:id="id.124" xmi:type="action:Calls" to="id.166" from="id.122"/>
    <actionRelation xmi:id="id.125" xmi:type="action:Writes" to="id.113" from="id.122"/>
    <actionRelation xmi:id="id.126" xmi:type="action:Flow"/>
  </codeElement>
  <actionRelation xmi:id="id.127" xmi:type="action:Flow" to="id.115" from="id.114"/>
</codeElement>
```

```
      <codeElement xmi:id="id.128" xmi:type="code:StorableUnit" name="recognizedDate" type="id.168"
                   kind="local"/>
      <codeElement xmi:id="id.129" xmi:type="action:ActionElement" name="c7" kind="MethodCall">
        <codeElement xmi:id="id.130" xmi:type="action:ActionElement" name="c7.1" kind="Call">
          <codeElement xmi:id="id.131" xmi:type="code:Value" name="&amp;quot;dateSigned&amp;quot;"/>
          <codeElement xmi:id="id.132" xmi:type="code:StorableUnit" name="t5" kind="register"/>
          <actionRelation xmi:id="id.133" xmi:type="action:Addresses" to="id.103" from="id.130"/>
          <actionRelation xmi:id="id.134" xmi:type="action:Calls" to="id.160" from="id.130"/>
          <actionRelation xmi:id="id.135" xmi:type="action:Writes" to="id.132" from="id.130"/>
          <actionRelation xmi:id="id.136" xmi:type="action:Flow" to="id.137" from="id.130"/>
        </codeElement>
        <codeElement xmi:id="id.137" xmi:type="action:ActionElement" name="c7.2" kind="New">
          <actionRelation xmi:id="id.138" xmi:type="action:Creates" to="id.168" from="id.137"/>
          <actionRelation xmi:id="id.139" xmi:type="action:Writes" to="id.128" from="id.137"/>
          <actionRelation xmi:id="id.140" xmi:type="action:Flow"/>
        </codeElement>
        <codeElement xmi:id="id.141" xmi:type="action:ActionElement" name="c7.3" kind="MethodCall">
          <actionRelation xmi:id="id.142" xmi:type="action:Reads" to="id.132" from="id.137"/>
          <actionRelation xmi:id="id.143" xmi:type="action:Calls" to="id.169" from="id.141"/>
          <actionRelation xmi:id="id.144" xmi:type="action:Writes" to="id.128" from="id.141"/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.145" xmi:type="code:Value"
           name="&amp;quot;SELECT * FROM contracts c, products p WHERE ID=? AND c.product=p.ID&amp;quot;"
           type="id.178"/>
    <codeElement xmi:id="id.146" xmi:type="code:MethodUnit" name="init" kind="constructor">
      <entryFlow xmi:id="id.147" to="id.148" from="id.146"/>
      <codeElement xmi:id="id.148" xmi:type="action:ActionElement" name="i1" kind="Assign">
        <actionRelation xmi:id="id.149" xmi:type="action:Reads" to="id.145" from="id.148"/>
        <actionRelation xmi:id="id.150" xmi:type="action:Writes" to="id.72" from="id.148"/>
      </codeElement>
    </codeElement>
  </codeElement>
</model>
<model xmi:id="id.151" xmi:type="code:CodeModel" name="Java packages">
  <codeElement xmi:id="id.152" xmi:type="code:Package" name="java.sql">
    <codeElement xmi:id="id.153" xmi:type="code:ClassUnit" name="DriverManager">
      <codeElement xmi:id="id.154" xmi:type="code:MethodUnit" name="getConnection" kind="abstract"/>
    </codeElement>
    <codeElement xmi:id="id.155" xmi:type="code:ClassUnit" name="Connection">
      <codeElement xmi:id="id.156" xmi:type="code:MethodUnit" name="prepareStatement" kind="abstract"/>
    </codeElement>
    <codeElement xmi:id="id.157" xmi:type="code:ClassUnit" name="ResultSet">
      <codeElement xmi:id="id.158" xmi:type="code:MethodUnit" name="next" kind="abstract"/>
      <codeElement xmi:id="id.159" xmi:type="code:MethodUnit" name="getBigDecimal" kind="abstract"/>
      <codeElement xmi:id="id.160" xmi:type="code:MethodUnit" name="getDate" kind="abstract"/>
    </codeElement>
    <codeElement xmi:id="id.161" xmi:type="code:ClassUnit" name="Statement">
      <codeElement xmi:id="id.162" xmi:type="code:MethodUnit" name="setLong" kind="abstract"/>
      <codeElement xmi:id="id.163" xmi:type="code:MethodUnit" name="executeQuery" kind="abstract"/>
    </codeElement>
  </codeElement>
  <codeElement xmi:id="id.164" xmi:type="code:Package" name="Money">
    <codeElement xmi:id="id.165" xmi:type="code:ClassUnit" name="Money">
```

```
        <codeElement xmi:id="id.166" xmi:type="code:MethodUnit" name="dollars" kind="abstract"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.167" xmi:type="code:Package" name="MfDate">
      <codeElement xmi:id="id.168" xmi:type="code:ClassUnit" name="MfDate">
        <codeElement xmi:id="id.169" xmi:type="code:MethodUnit" name="MfDate" kind="abstract"/>
      </codeElement>
    </codeElement>
  </model>
  <model xmi:id="id.170" xmi:type="code:CodeModel" name="Common Datatypes">
    <codeElement xmi:id="id.171" xmi:type="code:LanguageUnit" name="SQL datatypes">
      <codeElement xmi:id="id.172" xmi:type="code:IntegerType" name="sql int"/>
      <codeElement xmi:id="id.173" xmi:type="code:StringType" name="sql varchar"/>
      <codeElement xmi:id="id.174" xmi:type="code:DecimalType" name="sql decimal"/>
      <codeElement xmi:id="id.175" xmi:type="code:DateType" name="sql date"/>
      <codeElement xmi:id="id.176" xmi:type="code:BooleanType"/>
    </codeElement>
    <codeElement xmi:id="id.177" xmi:type="code:LanguageUnit" name="Java datatypes">
      <codeElement xmi:id="id.178" xmi:type="code:StringType"/>
      <codeElement xmi:id="id.179" xmi:type="code:IntegerType" name="java long"/>
      <codeElement xmi:id="id.180" xmi:type="code:IntegerType" name="java byte"/>
    </codeElement>
  </model>
</kdm:Segment>
```

18.10 StructuredData Class Diagram

The StructuredData class diagram provides basic meta-model constructs to define the XML files that can be used by enterprise applications for persistent storage or as an exchange mechanism between components. The class diagram shown in Figure 18.8 captures these classes and their relations.

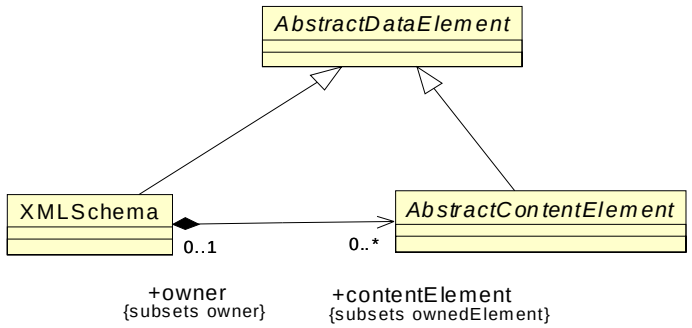

Figure 18.8 - StructuredData Class Diagram

18.10.1 XMLSchema

The XMLSchema class represents the top level container for a KDM metamodel of an XML document.

Superclass

AbstractDataElement

Associations

contentElement :AbstractContentElement[0..*]	Individual content elements owned by this schema.

Semantics

XMLSchema is a logical container for AbstractContentElements as well as some other DataResource elements (for example, DataEvents).

18.10.2 AbstractContentElement (abstract)

The AbstractContentElement class is an abstract parent for several concrete classes whose purpose is to represent the content of XML schemas and documents as well as various structured data items that can be associated with other KDM elements.

Superclass

AbstractDataElement

Semantics

AbstractContentElement represents common properties of content elements.

18.11 ContentElements Class Diagram

The ContentElements class diagram defines basic meta-model constructs to represent XML elements. The class diagram shown in Figure 18.9 captures these classes and their relations.

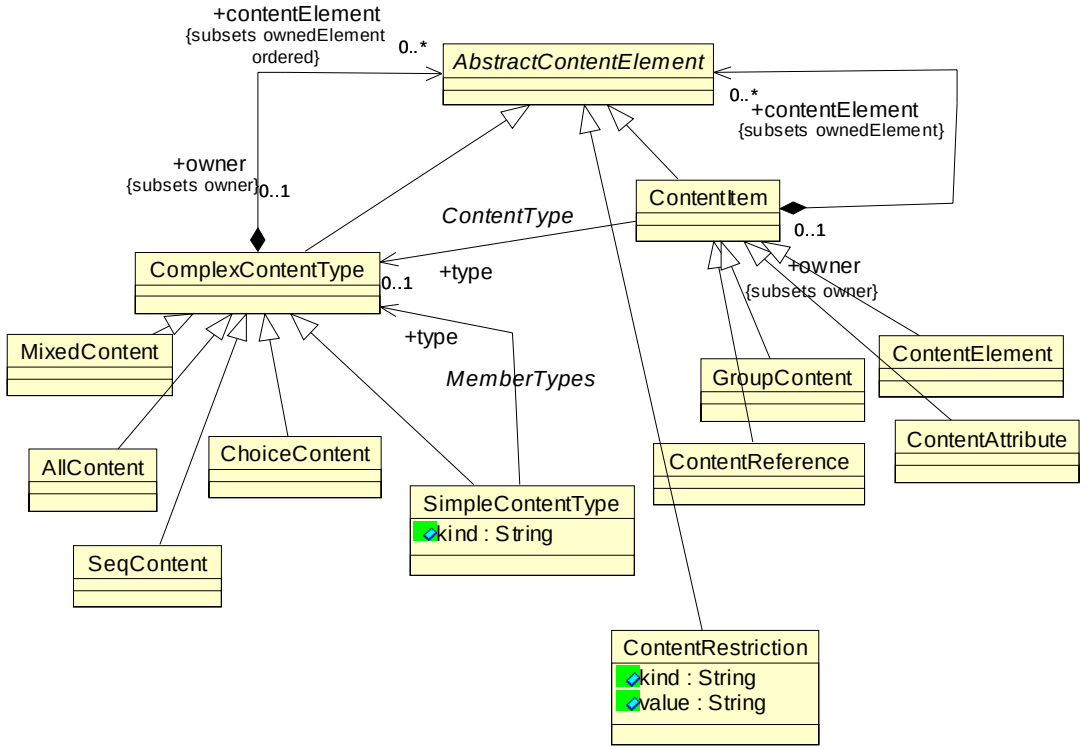

Figure 18.9 - ContentElements Class Diagram

18.11.1 ContentItem (generic)

The ContentItem class is a generic meta-model element that represents named items and references of the XML schema: elements, attributes, references, and groups.

Superclass

AbstractContentElement

Associations

contentElement :AbstractContentElement[0..*]	owned content elements
type:ComplexContentType[0..1]	content type of the current ContentItem

Semantics

18.11.2 ComplexContentType

The ComplexContentType class represents Complex Types of an XML schema definition. XSD indicators are modeled as subclasses of ComplexContentType.

Superclass

AbstractContentElement

Associations

contentElement :AbstractContentElement[0..*]	Owned content elements

Semantics

18.11.3 SimpleContentType

The SimpleContentType class represents Simple Types of an XML schema definition.

Superclass

ComplexContentType

Attributes

kind:String	content kind of the current SimpleContentType

Associations

type:ComplexContentType[0..*]	content type of the current ContentItem

Semantics

Simple types, such as string and decimal, are built in to XML Schema, while others are derived from the built-in's. The kind of SimpleContentType can be "list," "union," "enumeration," etc.

18.11.4 ContentRestriction

The ContentRestriction class represents restrictions to Simple Types, Elements, Attributes, and References.

Superclass

AbstractContentElement

Attributes

kind :String	type of the content restriction (XML)
value:String	value of the constraint

Semantics

kind is an XSD restriction, such as minExclusive, minInclusive, maxExclusive, maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whiteSpace, pattern; or XSD an element attribute, such as minOccurs, maxOccurs, required, fixed; or an XSD enumeration.

Example

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>

<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>

<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>

<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="XML Simple Content Example">
  <model xmi:id="id.0" xmi:type="data:DataModel">
    <dataElement xmi:id="id.1" xmi:type="data:XMLSchema" name="SimpleType examples">
      <contentElement xmi:id="id.2" xmi:type="data:SimpleContentType" name="MyInteger">
        <dataRelation xmi:id="id.3" xmi:type="data:RestrictionOf" to="id.27" from="id.2"/>
        <contentElement xmi:id="id.4" xmi:type="data:ContentRestriction"
                kind="minInclusive" value="10000"/>
        <contentElement xmi:id="id.5" xmi:type="data:ContentRestriction"
                kind="maxInclusive" value="99999"/>
```

```
    </contentElement>
    <contentElement xmi:id="id.6" xmi:type="data:SimpleContentType" name="SKU">
      <dataRelation xmi:id="id.7" xmi:type="data:RestrictionOf" to="id.29" from="id.2"/>
      <contentElement xmi:id="id.8" xmi:type="data:ContentRestriction"
          kind="pattern" value="&amp;quot;\d{3}-[A-Z]{2}&amp;quot;"/>
    </contentElement>
    <contentElement xmi:id="id.9" xmi:type="data:SimpleContentType" name="USState">
      <contentElement xmi:id="id.10" xmi:type="data:ContentRestriction"
               kind="enumeration" value="&amp;quot;AK&amp;quot;"/>
      <contentElement xmi:id="id.11" xmi:type="data:ContentRestriction"
               kind="enumeration" value="&amp;quot;AL&amp;quot;"/>
      <contentElement xmi:id="id.12" xmi:type="data:ContentRestriction"
               kind="enumeration" value="&amp;quot;AR&amp;quot;"/>
    </contentElement>
    <contentElement xmi:id="id.13" xmi:type="data:SimpleContentType" name="listOfMyIntType">
      <contentElement xmi:id="id.14" xmi:type="data:ListContent">
        <dataRelation xmi:id="id.15" xmi:type="data:TypedBy" to="id.2" from="id.14"/>
      </contentElement>
    </contentElement>
    <contentElement xmi:id="id.16" xmi:type="data:SimpleContentType" name="USStateList">
      <contentElement xmi:id="id.17" xmi:type="data:ListContent" name="">
        <dataRelation xmi:id="id.18" xmi:type="data:TypedBy" to="id.9" from="id.17"/>
      </contentElement>
    </contentElement>
    <contentElement xmi:id="id.19" xmi:type="data:SimpleContentType" name="SixUSStates">
      <dataRelation xmi:id="id.20" xmi:type="data:RestrictionOf" to="id.16" from="id.19"/>
      <contentElement xmi:id="id.21" xmi:type="data:ContentRestriction" kind="length" value="6"/>
    </contentElement>
    <contentElement xmi:id="id.22" xmi:type="data:SimpleContentType" name="zipUnion">
      <contentElement xmi:id="id.23" xmi:type="data:UnionContent">
        <dataRelation xmi:id="id.24" xmi:type="data:TypedBy" to="id.9" from="id.23"/>
        <dataRelation xmi:id="id.25" xmi:type="data:TypedBy" to="id.13" from="id.23"/>
      </contentElement>
    </contentElement>
  </dataElement>
  <dataElement xmi:id="id.26" xmi:type="data:XMLSchema" name="xsd">
    <contentElement xmi:id="id.27" xmi:type="data:SimpleContentType" name="xsd:Integer">
      <dataRelation xmi:id="id.28" xmi:type="data:DatatypeOf" to="id.41" from="id.27"/>
    </contentElement>
    <contentElement xmi:id="id.29" xmi:type="data:SimpleContentType" name="xsd:String">
      <dataRelation xmi:id="id.30" xmi:type="data:DatatypeOf" to="id.42" from="id.29"/>
    </contentElement>
    <contentElement xmi:id="id.31" xmi:type="data:SimpleContentType" name="xsd:Decimal">
      <dataRelation xmi:id="id.32" xmi:type="data:DatatypeOf" to="id.43" from="id.31"/>
    </contentElement>
    <contentElement xmi:id="id.33" xmi:type="data:SimpleContentType" name="xsd:positiveInteger">
      <dataRelation xmi:id="id.34" xmi:type="data:DatatypeOf" to="id.41" from="id.33"/>
    </contentElement>
    <contentElement xmi:id="id.35" xmi:type="data:SimpleContentType" name="xsd:date">
      <dataRelation xmi:id="id.36" xmi:type="data:DatatypeOf" to="id.44" from="id.35"/>
    </contentElement>
    <contentElement xmi:id="id.37" xmi:type="data:SimpleContentType" name="xsd:any"/>
    <contentElement xmi:id="id.38" xmi:type="data:SimpleContentType" name="xsd:NMTOKEN"/>
  </dataElement>
</model>
```

```
  <model xmi:id="id.39" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.40" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.41" xmi:type="code:IntegerType" name="xsd integer"/>
      <codeElement xmi:id="id.42" xmi:type="code:StringType" name="xsd string"/>
      <codeElement xmi:id="id.43" xmi:type="code:DecimalType" name="xsd decimal"/>
      <codeElement xmi:id="id.44" xmi:type="code:DateType" name="xsd date"/>
    </codeElement>
  </model>
</kdm:Segment>
```

18.11.5 AllContent Class

An AllContent class is a meta-model element that represents complex types with the “all” order indicator.

Superclass

ComplexContentType

Semantics

18.11.6 SeqContent Class

The SeqContent class is a meta-model element that represents complex types with the “sequence” order indicator.

Superclass

ComplexContentType

Semantics

18.11.7 ChoiceContent Class

A ChoiceContent class is a meta-model element that represents complex types with the “choice” order indicator.

Superclass

XMLComplexType

Semantics

18.11.8 GroupContent Class

A GroupContent class is a meta-model element that represents complex types with the “group” group indicator.

Superclass

ComplexContentType

Semantics

18.11.9 MixedContent Class

A MixedContent class is a meta-model element that represents complex types with the “mixed” indicator.

Superclass

ComplexContentType

Semantics

18.11.10 ContentAttribute Class

A ContentAttribute class is a meta-model element that represents the XML “attribute” declaration mechanism of XML Schemas.

Superclass

ContentItem

Semantics

18.11.11 ContentElement Class

A ContentElement class is a meta-model element that represents the XML “element” declaration mechanism of XML Schemas.

Superclass

ContentItem

Semantics

18.11.12 ContentReference Class

A ContentReference class is a meta-model element that represents the XML “reference” declaration mechanism of XML Schemas.

Superclass

ContentItem

Semantics

Example

```
<xsd:element name="letterBody">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="salutation">
        <xsd:complexType mixed="true">
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="quantity"    type="xsd:positiveInteger"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="shipDate"    type="xsd:date" minOccurs="0"/>
      <!-- etc. -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="USAddress" >
  <xsd:sequence>
    <xsd:element name="name"   type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city"   type="xsd:string"/>
    <xsd:element name="state"  type="xsd:string"/>
    <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice"  type="xsd:decimal"/>
          <xsd:element ref="comment"   minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="partNum" type="SKU" use="required"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
```

```
</xsd:complexType>

<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:attribute name="currency" type="xsd:string"/>
        <xsd:attribute name="value"    type="xsd:decimal"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group   ref="shipAndBill"/>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:group id="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>

<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
        xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
        xmlns:data="http://schema.omg.org/spec/KDM/1.2/data"
        xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm" name="XML Complex Content Example">
  <model xmi:id="id.0" xmi:type="data:DataModel">
    <dataElement xmi:id="id.1" xmi:type="data:XMLSchema" name="Complex Content">
      <contentElement xmi:id="id.2" xmi:type="data:ContentElement" name="letterBody">
        <dataRelation xmi:id="id.3" xmi:type="data:TypedBy" to="id.4" from="id.2"/>
        <contentElement xmi:id="id.4" xmi:type="data:MixedContent" name="m1">
          <contentElement xmi:id="id.5" xmi:type="data:SeqContent">
            <contentElement xmi:id="id.6" xmi:type="data:ContentElement" name="salutation">
              <dataRelation xmi:id="id.7" xmi:type="data:TypedBy" to="id.8" from="id.6"/>
              <contentElement xmi:id="id.8" xmi:type="data:MixedContent">
                <contentElement xmi:id="id.9" xmi:type="data:SeqContent">
                  <contentElement xmi:id="id.10" xmi:type="data:ContentElement" name="name">
                    <dataRelation xmi:id="id.11" xmi:type="data:TypedBy" to="id.88" from="id.10"/>
                  </contentElement>
                </contentElement>
              </contentElement>
            </contentElement>
            <contentElement xmi:id="id.12" xmi:type="data:ContentElement" name="quantity">
```

```
        <dataRelation xmi:id="id.13" xmi:type="data:TypedBy" to="id.92" from="id.12"/>
      </contentElement>
      <contentElement xmi:id="id.14" xmi:type="data:ContentElement" name="productName">
        <dataRelation xmi:id="id.15" xmi:type="data:TypedBy" to="id.88" from="id.14"/>
      </contentElement>
      <contentElement xmi:id="id.16" xmi:type="data:ContentElement" name="shipDate">
        <dataRelation xmi:id="id.17" xmi:type="data:TypedBy" to="id.94" from="id.16"/>
      </contentElement>
    </contentElement>
  </contentElement>
</contentElement>
<contentElement xmi:id="id.18" xmi:type="data:ComplexContentType" name="USAddress">
  <contentElement xmi:id="id.19" xmi:type="data:SeqContent">
    <contentElement xmi:id="id.20" xmi:type="data:ContentElement" name="name">
      <dataRelation xmi:id="id.21" xmi:type="data:TypedBy" to="id.88" from="id.20"/>
    </contentElement>
    <contentElement xmi:id="id.22" xmi:type="data:ContentElement" name="street">
      <dataRelation xmi:id="id.23" xmi:type="data:TypedBy" to="id.88" from="id.22"/>
    </contentElement>
    <contentElement xmi:id="id.24" xmi:type="data:ContentElement" name="city">
      <dataRelation xmi:id="id.25" xmi:type="data:TypedBy" to="id.88" from="id.24"/>
    </contentElement>
    <contentElement xmi:id="id.26" xmi:type="data:ContentElement" name="state">
      <dataRelation xmi:id="id.27" xmi:type="data:TypedBy" to="id.88" from="id.26"/>
    </contentElement>
    <contentElement xmi:id="id.28" xmi:type="data:ContentElement" name="zip">
      <dataRelation xmi:id="id.29" xmi:type="data:TypedBy" to="id.88" from="id.28"/>
    </contentElement>
  </contentElement>
  <contentElement xmi:id="id.30" xmi:type="data:ContentAttribute" name="country">
    <dataRelation xmi:id="id.31" xmi:type="data:TypedBy" to="id.97" from="id.30"/>
    <contentElement xmi:id="id.32" xmi:type="data:ContentRestriction"
          kind="fixed" value="&amp;quot;US&amp;quot;"/>
  </contentElement>
</contentElement>
<contentElement xmi:id="id.33" xmi:type="data:ComplexContentType" name="items">
  <contentElement xmi:id="id.34" xmi:type="data:SeqContent">
    <contentElement xmi:id="id.35" xmi:type="data:ContentElement" name="item">
      <dataRelation xmi:id="id.36" xmi:type="data:TypedBy" to="id.39" from="id.35"/>
     <contentElement xmi:id="id.37" xmi:type="data:ContentRestriction" kind="minOccurs" value="0"/>
      <contentElement xmi:id="id.38" xmi:type="data:ContentRestriction"
              kind="maxOccurs" value="unbounded"/>
      <contentElement xmi:id="id.39" xmi:type="data:ComplexContentType" name="i">
        <contentElement xmi:id="id.40" xmi:type="data:SeqContent">
          <contentElement xmi:id="id.41" xmi:type="data:ContentElement" name="productName1">
            <dataRelation xmi:id="id.42" xmi:type="data:TypedBy" to="id.88" from="id.41"/>
          </contentElement>
          <contentElement xmi:id="id.43" xmi:type="data:ContentElement" name="quantity1">
            <dataRelation xmi:id="id.44" xmi:type="data:TypedBy" to="id.45" from="id.43"/>
            <contentElement xmi:id="id.45" xmi:type="data:SimpleContentType" name="st1">
              <dataRelation xmi:id="id.46" xmi:type="data:RestrictionOf" to="id.92" from="id.45"/>
              <contentElement xmi:id="id.47" xmi:type="data:ContentRestriction"
                        kind="maxExclusive" value="100"/>
            </contentElement>
          </contentElement>
```

```
            <contentElement xmi:id="id.48" xmi:type="data:ContentElement" name="USPrice">
              <dataRelation xmi:id="id.49" xmi:type="data:TypedBy" to="id.90" from="id.48"/>
            </contentElement>
            <contentElement xmi:id="id.50" xmi:type="data:ContentReference">
              <dataRelation xmi:id="id.51" xmi:type="data:ReferenceTo" to="id.83" from="id.50"/>
              <contentElement xmi:id="id.52" xmi:type="data:ContentRestriction"
                   kind="minOccurs" value="0"/>
            </contentElement>
            <contentElement xmi:id="id.53" xmi:type="data:ContentElement" name="shipDate1">
              <dataRelation xmi:id="id.54" xmi:type="data:TypedBy" to="id.94" from="id.53"/>
            </contentElement>
          </contentElement>
          <contentElement xmi:id="id.55" xmi:type="data:ContentAttribute" name="partNum">
            <dataRelation xmi:id="id.56" xmi:type="data:TypedBy" from="id.55"/>
            <contentElement xmi:id="id.57" xmi:type="data:ContentRestriction"
                   kind="use" value="required"/>
          </contentElement>
        </contentElement>
      </contentElement>
    </contentElement>
  </contentElement>
  <contentElement xmi:id="id.58" xmi:type="data:ContentElement" name="international price">
    <contentElement xmi:id="id.59" xmi:type="data:ComplexContentType" name="">
      <dataRelation xmi:id="id.60" xmi:type="data:RestrictionOf" to="id.96" from="id.59"/>
      <contentElement xmi:id="id.61" xmi:type="data:ContentAttribute" name="currency1">
        <dataRelation xmi:id="id.62" xmi:type="data:TypedBy" to="id.88" from="id.61"/>
      </contentElement>
      <contentElement xmi:id="id.63" xmi:type="data:ContentAttribute" name="value">
        <dataRelation xmi:id="id.64" xmi:type="data:TypedBy" to="id.90" from="id.61"/>
      </contentElement>
    </contentElement>
  </contentElement>
  <contentElement xmi:id="id.65" xmi:type="data:ComplexContentType" name="PurchaseOrderType">
    <contentElement xmi:id="id.66" xmi:type="data:SeqContent">
      <contentElement xmi:id="id.67" xmi:type="data:ChoiceContent">
        <contentElement xmi:id="id.68" xmi:type="data:ContentReference">
          <dataRelation xmi:id="id.69" xmi:type="data:ReferenceTo" to="id.79" from="id.68"/>
        </contentElement>
        <contentElement xmi:id="id.70" xmi:type="data:ContentElement" name="singleUSAddress">
          <dataRelation xmi:id="id.71" xmi:type="data:TypedBy" to="id.18" from="id.70"/>
        </contentElement>
      </contentElement>
      <contentElement xmi:id="id.72" xmi:type="data:ContentReference">
        <dataRelation xmi:id="id.73" xmi:type="data:ReferenceTo" to="id.83" from="id.72"/>
       <contentElement xmi:id="id.74" xmi:type="data:ContentRestriction" kind="minOccurs" value="0"/>
      </contentElement>
      <contentElement xmi:id="id.75" xmi:type="data:ContentElement" name="items">
        <dataRelation xmi:id="id.76" xmi:type="data:TypedBy" to="id.33" from="id.75"/>
      </contentElement>
    </contentElement>
    <contentElement xmi:id="id.77" xmi:type="data:ContentAttribute" name="orderDate">
      <dataRelation xmi:id="id.78" xmi:type="data:TypedBy" to="id.94" from="id.77"/>
    </contentElement>
  </contentElement>
  <contentElement xmi:id="id.79" xmi:type="data:GroupContent" name="shipAndBill">
```

```
        <contentElement xmi:id="id.80" xmi:type="data:SeqContent">
          <contentElement xmi:id="id.81" xmi:type="data:ContentElement"/>
          <contentElement xmi:id="id.82" xmi:type="data:ContentElement"/>
        </contentElement>
      </contentElement>
      <contentElement xmi:id="id.83" xmi:type="data:ContentElement" name="comment">
        <dataRelation xmi:id="id.84" xmi:type="data:TypedBy" to="id.88" from="id.83"/>
      </contentElement>
    </dataElement>
    <dataElement xmi:id="id.85" xmi:type="data:XMLSchema" name="xsd">
      <contentElement xmi:id="id.86" xmi:type="data:SimpleContentType" name="xsd:Integer">
        <dataRelation xmi:id="id.87" xmi:type="data:DatatypeOf" to="id.100" from="id.86"/>
      </contentElement>
      <contentElement xmi:id="id.88" xmi:type="data:SimpleContentType" name="xsd:String">
        <dataRelation xmi:id="id.89" xmi:type="data:DatatypeOf" to="id.101" from="id.88"/>
      </contentElement>
      <contentElement xmi:id="id.90" xmi:type="data:SimpleContentType" name="xsd:Decimal">
        <dataRelation xmi:id="id.91" xmi:type="data:DatatypeOf" to="id.102" from="id.90"/>
      </contentElement>
      <contentElement xmi:id="id.92" xmi:type="data:SimpleContentType" name="xsd:positiveInteger">
        <dataRelation xmi:id="id.93" xmi:type="data:DatatypeOf" to="id.100" from="id.92"/>
      </contentElement>
      <contentElement xmi:id="id.94" xmi:type="data:SimpleContentType" name="xsd:date">
        <dataRelation xmi:id="id.95" xmi:type="data:DatatypeOf" to="id.103" from="id.94"/>
      </contentElement>
      <contentElement xmi:id="id.96" xmi:type="data:SimpleContentType" name="xsd:any"/>
      <contentElement xmi:id="id.97" xmi:type="data:SimpleContentType" name="xsd:NMTOKEN"/>
    </dataElement>
  </model>
  <model xmi:id="id.98" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.99" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.100" xmi:type="code:IntegerType" name="xsd integer"/>
      <codeElement xmi:id="id.101" xmi:type="code:StringType" name="xsd string"/>
      <codeElement xmi:id="id.102" xmi:type="code:DecimalType" name="xsd decimal"/>
      <codeElement xmi:id="id.103" xmi:type="code:DateType" name="xsd date"/>
    </codeElement>
  </model>
</kdm:Segment>
```

18.12 ContentRelations Class Diagram

The ContentRelations class diagram provides basic meta-model relationships that represent various structural properties of the content. The class diagram shown in Figure 18.10 captures these classes and their relations.

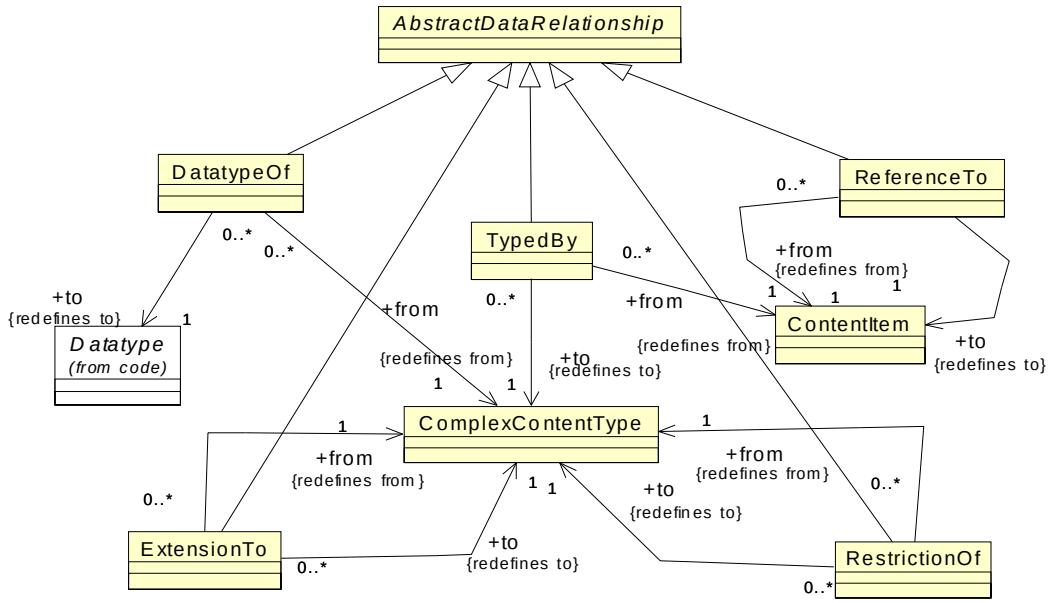

Figure 18.10 - ContentRelations Class Diagram

18.12.1 TypedBy Class

The TypedBy class represents the relationship between a ContentItem and a content type, that can be represented by a ComplexContentType class or one of its subclasses.

Superclass

AbstractDataRelationship

Associations

from:ContentItem[1]	the content element or attribute
to:ComplexContentType[1]	the content type element

Constraints

1. The “from” endpoint should be a ContentElement or a ContentAttribute class.

Semantics

TypedBy relationship represents an association between a content element and its type when this type is user-defined. This relationship is similar to HasType from CodeModel.

18.12.2 DatatypeOf Class

The DatatypeOf class represents the relationship between a CompelxContentType and a Datatype defined in some Code model.

Superclass

AbstractDataRelationship

Associations

from:ComplexContentType[1]	the content type
to:Datatype[1]	the datatype element

Semantics

DatatypeOf relationship represents an association between a content type and primitive datatype.

18.12.3 ReferenceTo Class

The ReferenceTo class represents the relationship between a ContentReference and a ContentElement, ContentAttribute, or ContentGroup definition.

Superclass

AbstractDataRelationship

Associations

from:ContentItem[1]	the content reference
to:ContentItem[1]	the content element or attribute or group

Constraints

1. The “from” endpoint should be a ContentReference.
2. The “to” endpoint should be a ContentElement, a ContentAttribute, or GroupContent.

Semantics

ReferenceTo relationship represents an association between a content reference and the corresponding definition.

18.12.4 ExtensionTo Class

The ExtensionTo class represents the relationship between two content types, where one type is an extension to another. The semantics of deriving new types by extension is that as the result a new complex type or simple type is defined that contains all the elements of the original type plus additional elements that are provided as the extension.

Superclass

AbstractDataRelationship

Associations

from:ComplexContentType[1]	the new (extended) content type
to:ComplexContentType[1]	the base content type

Constraints

Semantics

ExtensionTo relationship represents an association between a content type and its base type.

18.12.5 RestrictionOf Class

The RestrictionOf class represents the relationship between two content types, where one type is a restriction to another. The semantics of deriving new types by restriction is that as the result a new complex type or simple type is defined that contains all the elements and constraints of the original type plus additional constraints that are provided as the restriction.

Superclass

AbstractDataRelationship

Associations

from:ComplexContentType[1]	the new (restricted) content type
to:ComplexContentType[1]	the base content type

Semantics

RestrictionOf relationship represents an association between a content type and its base type.

18.13 ExtenededDataElements Class Diagram

The ExtendedDataElements class diagram defines two “wildcard” generic elements for the data model as determined by the KDM model pattern: a generic data entity and a generic data relationship..

The classes and associations of the ExtendedDataElements diagram are shown in Figure 18.11.

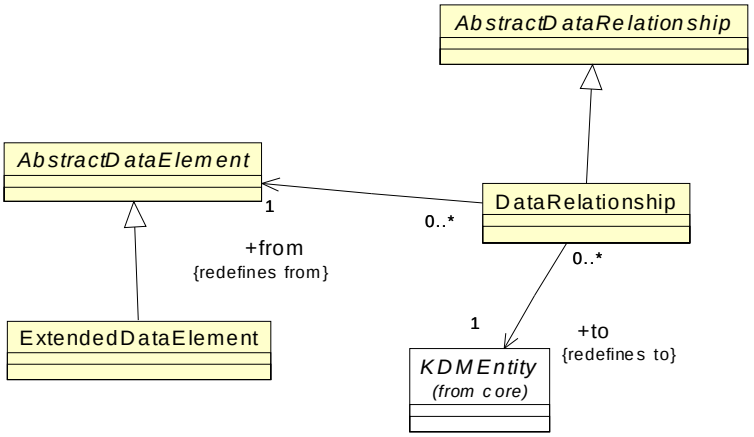

Figure 18.11 - ExtendedDataElements Class Diagram

18.13.1 ExtendedDataElement Class

The ExtendedDataElement class is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractDataElement

Constraints

1. ExtendedDataElement should have at least one stereotype.

Semantics

A data entity with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model entity type of the data model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

18.13.2 DataRelationship Class

The DataRelationship class is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractDataRelationship

Associations

from:AbstractDataElement[1]	the data element origin endpoint of the relationship
to:KDMEntity[1]	the target of the relationship

Constraints

1. DataRelationship should have at least one stereotype.

Semantics

A data relationship with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model relationship types of the data model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

Part IV - Abstractions Layer

Abstraction Layer defines a set of meta-model elements that represent domain-specific and application-specific abstractions, as well as artifacts related to the build process of the exsting software system. The meta-model elements of the Abstractions Layer provide various containers and groups to other meta-model elements.

Abstractions Layer defines the following 3 KDM Models:

- Structure
- Conceptual
- Build

19 Structure Package

19.1 Overview

Structure package defines meta-model elements that represent architectural components of existing software systems, such as subsystems, layers, packages, etc. and define traceability of these elements to other KDM facts for the same system.

The Structure package defines an *architecture viewpoint* for the Structure domain. The architectural views based on the viewpoint defined by the Structure model represent how the structural elements of the software system are related to the modules defined in the Code views that correspond to the Code architectural viewpoint defined by the Code model. The architectural viewpoint is defined as follows.

- **Concerns**:
  - What are the structural elements of the system, and what is the organization of these elements?
  - What software elements compose the system?
  - How the structural elements of the system are related to the computational elements?
  - What are the connections of these elements based on the relationships between the corresponding computational elements?
  - What are the interfaces of the structural elements of the system?
- **Viewpoint language**:

  Structure views conform to KDM XMI schema. The *viewpoint language* for the Structure *architectural viewpoint* is defined by the Structure package. It includes abstract entity AbstractStructureElement, and several concrete entities, such as Subsystem, Layer, Component, SoftwareSystem, ArchitectureView. The viewpoint language for the Structure architectural viewpoint also includes an abstract relationship AbstractStructureRelationship.
- **Analytic methods:**

  The Structure *architectural viewpoint* supports the following main kinds of checking:
  - Attachment (are components properly connected?)
  - Coupling and cohesion (the number of internal relationship within a component compared to the number of relationships to other components)
  - Efferent and afferent relationship (uses of a component by other components and usages of other component by the given component)
  - Interfaces (what is the required and provided interface of the given component)

  Structure Views are used in combination with Code views, Data views, Platform views, UI views and Inventory views. Specifically, Structure views corresponding to this architectural viewpoint represent how the structural elements of the software system are related to the modules defined in the Code views that correspond to the Code architectural viewpoint, defined by the Code package.
- **Construction methods**:
  - Structure views that correspond to the KDM Structure architectural viewpoint are usually constructed by

analyzing architecture models of the given system. The Structure extractor tool uses the knowledge of the architecture models to produce one or mode Structure views as output

- As an alternative, structure views can be produced manually using the input from the architect of the system and architecture documentation
- Construction of the Structure view is determined by the architectural description of the system
- Construction of the Structure views corresponding to a particular architectural description may involve additional information (system-specific or architecture-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations

The organization of the system may be presented as a single Structure view or a set of multiple Structure view showing layers, components, subsystems, or packages. The reach of this representation extends from a uniform architecture to entire family of module-sharing subsystems.

The Structure model owns a collection of StructuralElement instances.

Packages are the leaf elements of the Structure model, representing a division of a system's Code Modules into discrete, non-overlapping parts. An undifferentiated architecture is represented by a single Package.

StructuralGroup recursively gathers StructuralElements to represent various architectural divisions. The Software System subclass provides a gathering point for all the system's packages directly or indirectly through other Structure elements. The packages may be further grouped into Subsystems, Layers, and Components, or Architecture Views.

19.2 Organization of the Structure Package

The Structure package defines a collection of meta-model elements whose purpose is to represent architectural organization of the existing software system.

The Structure package consists of the following 3 class diagrams:

- StructureModel
- StructureInheritances
- ExtendedStructureElements

The Structure package depends on the following packages:

- Core
- kdm

19.3 StructureModel Class Diagram

The StructureModel class diagram follows the uniform pattern for KDM models and extends the KDM framework with specific meta-model elements related to high-level structural elements and their associations. The class diagram shown in Figure 19.1 captures these classes and their relations.

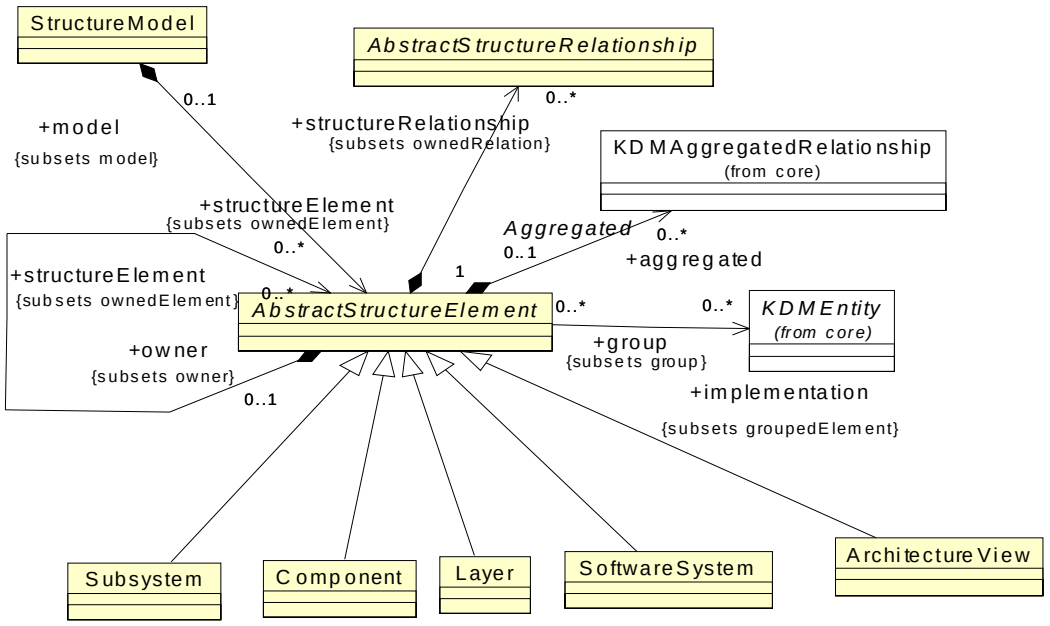

Figure 19.1 - StructureModel Class Diagram

19.3.1 StructureModel Class

The StructureModel is a specific KDM model that represents the logical organization of a software system and owns all of the system's StructuralElements.

Superclass

KDMModel

Associations

structureElement:AbstractStructureElement[0..*]	structure elements owned by the model

Semantics

19.3.2 AbstractStructureElement Class (abstract)

The AbstractStructureElement represents an architectural part, related to the organization of the existing software system into modules.

Superclass

KDMEntity

Associations

structureElement:AbstractStructureElement[0..*]	structure elements owned by the model
structureRelationship:AbstractStructureRelationship[0..*]	
aggregated:KDMAggregatedRelationship[0..*]	
implementation:KDMEntity[0..*]	

Semantics

19.3.3 AbstractStructureRelationship Class (abstract)

The AbstractStructureRelationship class

Superclass

KDMRelationship

Semantics

19.3.4 Subsystem Class

The Subsystem collects the architectural parts of a software subsystem. The parts may be any other StructuralElement.

Superclass

StructureGroup

Semantics

19.3.5 Layer Class

The Layer collects the architectural parts of a software subsystem to represent a software layer. The parts may be any other StructuralElement.

Superclass

StructureGroup

Semantics

19.3.6 Component Class

The Component represents a collection, directly or indirectly, of code resources, which comprises an architectural component.

Superclass

StructureGroup

Semantics

19.3.7 SoftwareSystem Class

The SoftwareSystem represents the entire system organization. It may contain subsystem or other StructureElements.

Superclass

StructureGroup

Semantics

19.3.8 ArchitectureView Class

The ArchitectureView class represents an arbitrary *architectural view*, as defined by ISO 42010. Within a KDM instance an ArchitectureView element may be used in a Structure model either stand-alone or in combination with other elements defined by the Structure package. The KDM ArchitectureView own a collection of KDM entities that corresponds to a particular architectural view of the software system. To conform to the ISO 42010 requirements for architectural description, the creator of the KDM instance should further document the corresponding *architectural viewpoint* by using a stereotype to the ArchitectureView element, attributes or annotations).

Superclass

StructureGroup

Semantics

19.4 StructureInheritances Class Diagram

The StructureInheritances class diagram shown in Figure 19.2 depicts how various data classes are types of Core KDM classes. Each of the Structure Package classes within this diagram inherits certain properties from KDM classes defined within the KDM Core Package.

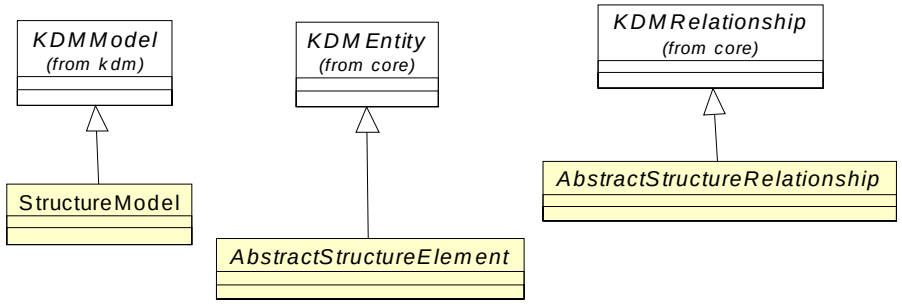

Figure 19.2 - StructureInheritances Class Diagram

19.5 ExtendedStructureElements Class Diagram

The ExtendedStructureElements class diagram defines two “wildcard” generic elements for the structure model as determined by the KDM model pattern: a generic structure entity and a generic structure relationship.

The classes and associations of the ExtendedStructureElements diagram are shown in Figure 19.3.

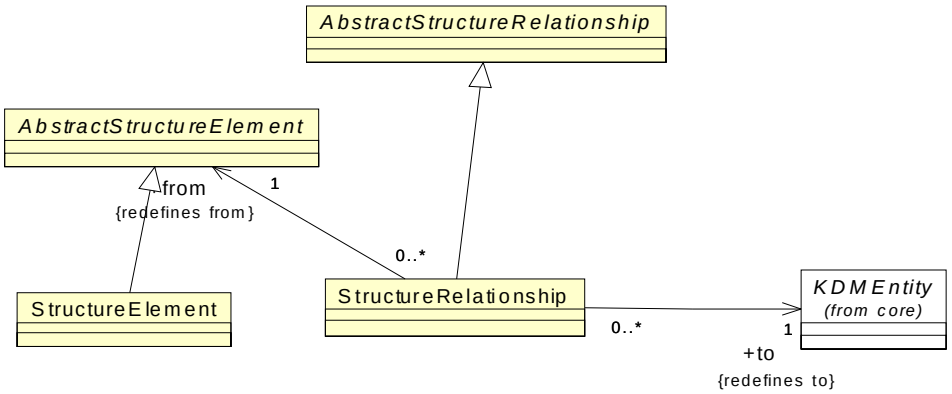

Figure 19.3 - ExtendedStructureElements Class Diagram

19.5.1 StructureElement Class (generic)

The StructureElement class is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractStructureElement

Constraints

1. StructureElement should have at least one stereotype.

Semantics

A structure entity with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity type of the structure model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

19.5.2 StructureRelationship Class (generic)

The StructureRelationship class is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractStructureRelationship

Associations

from:AbstractStructureElement[1]	the structure element origin endpoint of the relationship
to:KDMEntity[1]	the target of the relationship

Constraints

1. StructureRelationship should have at least one stereotype.

Semantics

A structure relationship with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model relationship type of the structure model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

20 Conceptual Package

20.1 Overview

The Conceptual Model defined in the KDM Conceptual package provides constructs for creating a conceptual model during the analysis phase of knowledge discovery from existing code.

The Conceptual package defines an *architectural viewpoint* for the Business Rules domain.

- **Concerns**:
  - What are the domain terms implemented by the system?
  - What are the behavior elements of the system?
  - What are the business rules implemented by the system?
  - What are the scenarios supported by the system?
- **Viewpoint language**:

  Conceptual views conform to KDM XMI schema. The *viewpoint language* for the Conceptual *architectural viewpoint* is defined by the Conceptual package. It includes abstract entity AbstractConceptualElement, and several concrete entities, such as TermUnit, FactUnit, RuleUnit, Scenario, BehaviorUnit. The viewpoint language for the Conceptual architectural viewpoint also includes ConceptualFlow relationship, which is a subclass of an abstract relationship AbstractConceptualRelationship.
- **Analytic methods**

  The Conceptual *architectural viewpoint* supports the following main kinds of checking:
  - Conceptual relationships (what are the relationships between conceptual entities, based on their implementation by the Code and Data entities?)
  - Scenario flow (what are the control flow relationship between the two scenarios based on the flow between action elements referenced by each scenario).
  - BehaviorUnit coupling (what are the control flow and data flow relationships between two behavior units based on the action elements referenced by each behavior unit).
  - Business Rule analysis (what is the logic of the business rule based on the action elements referenced by the business rule).

  Conceptual Views are used in combination with Code views, Data views, Platform views, UI views, and Inventory views.
- Construction methods:
  - Conceptual views can be produced manually using the input from the information analysis and the architect of the system and architecture documentation.
  - Construction of the Conceptual view is determined by the domain model and the architectural description of the system.

- Construction of the Conceptual views corresponding to a particular architectural description may involve additional information (system-specific or architecture-specific). This information can be attached to KDM elements using stereotypes, attributes or annotations.

The Conceptual Model enables mapping of KDM compliant model to models compliant to other specifications. Currently, it provides “concept” classes - TermUnit and FactUnit facilitating mapping to SBVR.

These meta-model elements support the process of mining higher level (conceptual) information from lower-level KDM models and capturing it as additional KDM entities and relationship, thus allowing further analysis of an enriched model. KDM Conceptual model is aligned with SBVR specification in the following way. The KDM Conceptual Model allows representing three “concepts” that are key to SBVR: Term, Fact, and Rule.

The following is a mapping of these KDM “concepts” to the SBVR terminology:

- Term corresponds to SBVR Noun (collectively referring to SBVR Terms and SBVR Names)
- Fact corresponds to SBVR Fact
- Rule represents a condition, group of conditions, or constraint.

The SBVR “concepts” (i.e., Term, Fact, and Rule) are not defined in KDM. Instead, the KDM Conceptual Model defines the implementations of these “concepts” - TermUnit, FactUnit, and RuleUnit. The mapping between KDM and SBVR is facilitated with the help of (0..*) to (0..*) relationships between pairs (i.e., <Term, TermUnit> and <Fact, FactUnit> and <Rule, RuleUnit>) as shown in Figure 20.1.

The ConceptualModel also provides “behavior” types - BehaviorUnit and ScenarioUnit that support mapping to various external models including but not limited to activities/flow chart and swim lane diagrams, and use case scenarios. The following explains the difference between these “behavior” types:

- BehaviorUnit represents a behavior graph with several paths through the application logic and associated conditions. The “implementation” of this graph is provided by the ActionElements connected with Flow relations, from the Program Elements KDM layer. The graph can be as small as a single ActionElement. BehaviorUnit is an “abstraction” of ActionElements since it provides a modeling element for representing a collection of ActionElements that is meaningful from the application domain perspective, and further manipulate with this representation as a first class citizen of the ConceptualModel of KDM.
- ScenarioUnit represents a path (or multiple related paths) through the behavior graph. For example, ScenarioUnit corresponds to a trace through the systems, or a “use case.” ScenarioUnit can own an entire collection of BehaviorUnits, connected with ConceptualFlow elements and can thus represent a slice of the original behavior graph in the implementation of the software system. The conditions responsible for navigation between alternative paths within the graph can be represented as RuleUnits.
- RuleUnit represents a condition, a group of conditions, or a constraint. RuleUnit is a representation for some meaningful navigation conditions within behavior graphs represented by several BehaviorUnits.

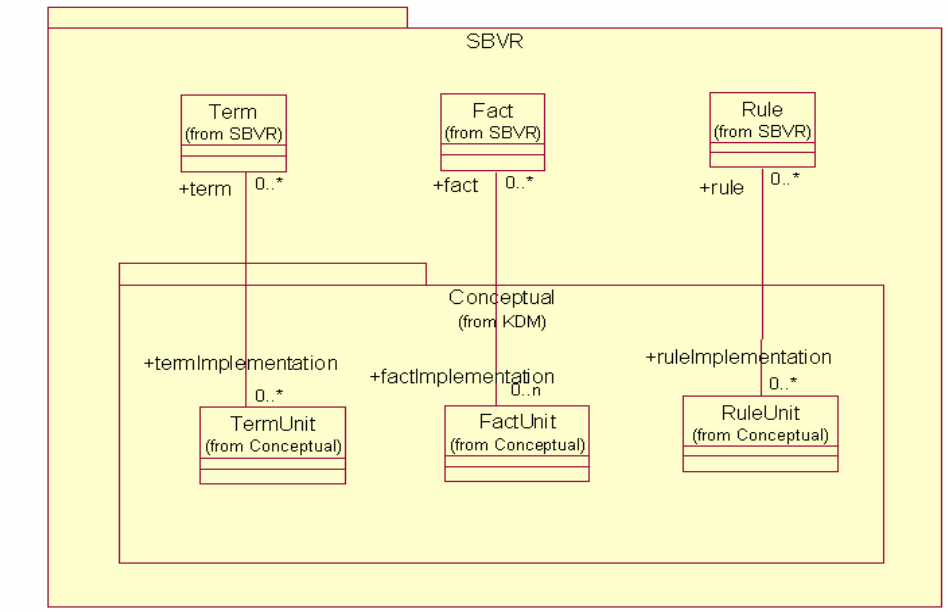

Figure 20.1 - Mapping between KDM and SBVR

20.2 Organization of the Conceptual Package

The Conceptual package defines meta-model elements that represent high-level, high-value application-specific "conceptual" elements of existing software systems and their traceability to other KDM facts.

The Conceptual Package consists of the following 5 class diagrams.

1. ConceptualModel
2. ConceptualInheritances
3. ConceptualElements
4. ConceptualRelations
5. ExtendedConceptualElements

The Conceptual package depends on the following packages:

- Core
- kdm

20.3 ConceptualModel Class Diagram

The ConceptualModel class diagram collects together all classes and associations of the Conceptual package. They provide basic meta-model constructs to define specialized "concept" types. These meta-model objects and relationships between them will be used as a foundation for a conceptual model built by a mining tool as a result of knowledge discovery from existing code. The ConceptualModel diagram defines meta-model elements to represent application-

specific “concept” types, “fact” types, and “rules.” An example of a “concept” is a “customer,” or a “savings account.” An example of a “fact” is a “customer opens a new savings account.” An example of a “rule” is “if the initial amount of money in a saving account is greater than $1000.00, then offer a higher interest account.” Even in a well-designed system an application-domain specific concept, like “customer” is rarely mapped one-to-one to a single programming language construct that can be directly discovered in the source code of the existing system. Instead, such “concept” is implemented by multiple programming language constructs, often spanning multiple source files, programming languages, and technologies. These KDM meta-model objects and relationships between them provide the foundation for mining business rules and other high-value application-specific knowledge from existing code.

ConceptualModel follows the regular KDM pattern of extending the common Infrastructure framework by defining a specific KDM model class, an abstract superclass of all modeling elements in the ConceptualModel - the AbstractConceptualElement class. ConceptualModel provides another abstract superclass for all relationships, specific to this model - AbstractConceptualRelationship class. All meta-model elements of the ConceptualModel extend the AbstractConceptualElement class and implement the “model” and “ownedRelation” properties. Each entity of the ConceptualModel can own relationships, specific to this model. According to this pattern, each entity should own relationships that originate from this entity (i.e., the value of the “from” property should be equal to the id of the owner of the relationship). This framework provides several extension points for the KDM light-weight extension mechanism. In particular, in accordance to the general KDM pattern, the ConceptualModel framework includes a generic extensible modeling element ConceptualElement, and a generic ConceptualRelationship class.

The class diagram shown in Figure 20.2 captures these classes and their relations.

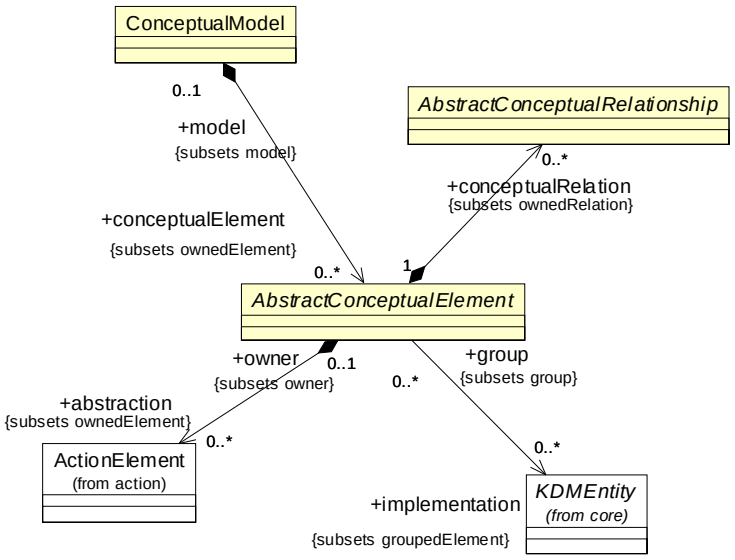

Figure 20.2 - ConceptualModel Class Diagram

20.3.1 ConceptualModel

The ConceptualModel class is a specific KDM model that owns collections of facts about conceptual elements implemented by a given existing software system.

Superclass

KDMModel

Associations

conceptualElement:AbstractConceptualElement[0..*]	Identifies the root “concept” elements of the hierarchy of the conceptual elements contained in the model. The ConceptualModel can contain zero or more such trees.

Semantics

20.3.2 AbstractConceptualElement (abstract)

AbstractConceptualElement class is the top superclass for the ConceptualModel. It defines several common properties for all further meta-model elements in the Conceptual Package. In particular, it defines the fundamental “implementation” property - a KDM grouping mechanism to link conceptual elements to the implementation elements. The “implementation” property represents the set of elements in lower-level KDM model that represents implementation-specific high-fidelity knowledge of the software system. The “implementation” property realizes the mapping between the implementation of a certain concept to the KDM entity representing this concept - some concrete subclass of the AbstractConceptualElement. The set of KDM entities available through the “implementation” property becomes the “extent” of the application-specific concept being represented by the conceptual element. The conceptual element itself becomes the “handle” for the otherwise intangible and abstract high-value application domain specific concept.

It is expected that building conceptual models in general, and especially, determining the appropriate “implementation set,” is a difficult value-added knowledge discovery process that may involve domain experts and application experts. KDM framework provides the intermediate representation for capturing the knowledge generated by this process.

Superclass

KDMEntity

Associations

conceptualRelation:AbstractConceptualRelation[0..*]	For each concrete instance of AbstractConceptualElement this property represents the set of conceptual relationships that originate from this element.
implementation:KDMEntity[0..*]	For each concrete instance of AbstractConceptualElement this property represents the set of KDM entities that realize the high-level concept in the low-level artifacts of the existing system.
abstraction:ActionElement[0..*]	This element represents action elements that are owned by the conceptual element and that represent semantic associations for the conceptual element.
source:SourceRef[0..*]	Traceability links to the physical artifacts represented by this element.

Constraints

1. For each conceptual element, the value of the from property of each conceptual relationship, owned by this element, should be equal to the identity of this element (the “relationship encapsulation” pattern).

20.3.3 AbstractConceptualRelationship Class (abstract)

The AbstractConceptualRelationship class is determined by the KDM model pattern. It provides a common superclass for specific KDM relationships, defined in the Conceptual package.

Semantics

20.4 ConceptualInheritances Class Diagram

The ConceptualInheritance class diagram defines how the conceptual meta-model elements fit into the KDM Infrastructure. The ConceptualInheritances class diagram is shown in Figure 20.3.

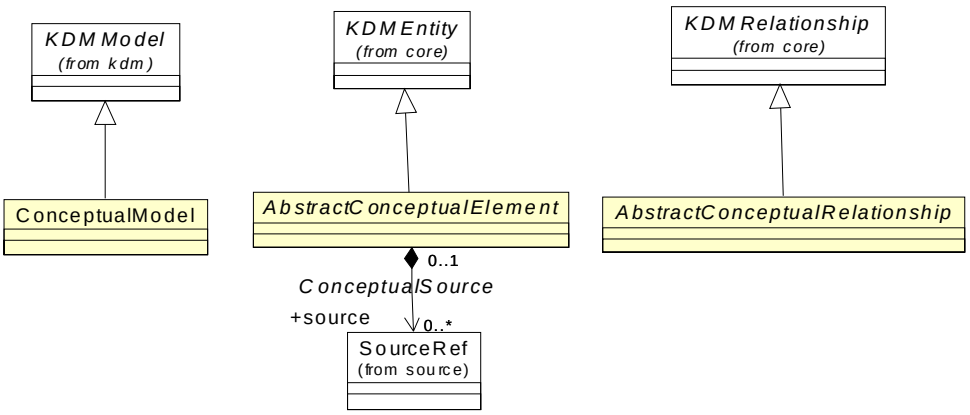

Figure 20.3 - ConceptualInheritances Class Diagram

20.5 ConceptualElements Class Diagram

ConceptualElements class diagram defines specific KDM modeling elements for representing domain-specific concepts as they are implemented by existing software systems. These elements are concrete subclasses of the AbstractConceptualElement class.

The classes and association of the ConceptualElements class diagram are shown at Figure 20.4.

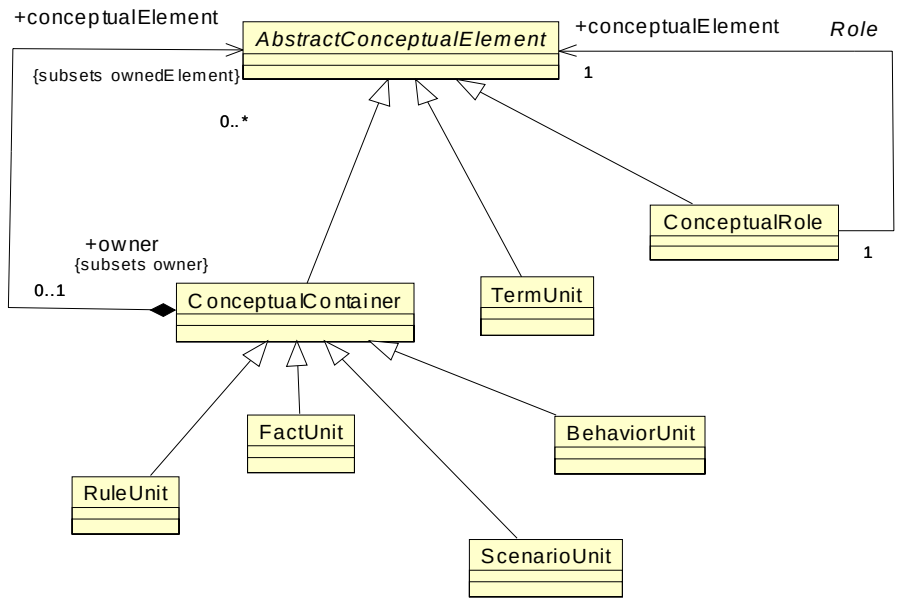

Figure 20.4 - ConceptualElements Class Diagram

20.5.1 ConceptualContainer Class

The ConceptualContainer class is a generic meta-model element that represents a container for conceptual entities. Several other concrete conceptual elements are subclasses of ConceptualContainer, so that they can also own other conceptual elements. The purpose of the ConceptualContainer meta-model element is to facilitate hierarchical organization and grouping of “concepts” within Conceptual Model. ConceptualContainer also can be used as an extended modeling element with a stereotype.

Superclass

AbstractConceptualElement

Associations

conceptualElement:AbstractConceptualElement[0..*]	elements that are owned by this container

Constraints

1. ConceptualUnit should not own ConceptualRole elements.

20.5.2 TermUnit

The TermUnit class represents an arbitrary collection of KDM entities from the Program Elements layer or PlatformResource layer. From the knowledge discovery perspective, such collection may include program elements involved in the implementation of a certain application domain concept. A TermUnit then provides a representation of

such concept inside the KDM model, which can be used for further analysis and later exported into a business rule modeling tool in the process known as application business rules mining. The TermUnit class is aligned with SBVR term or name concepts. Semantically, a TermUnit represents some “noun concept.”

Superclass

AbstractConceptualElement

Semantics

20.5.3 FactUnit

The FactUnit class represents an association between multiple Conceptual entities, such as TermUnit or FactUnit. This association can be unary, binary, or n-ary. From the knowledge mining perspective, a FactUnit may correspond to some behavior of the software system (for example, a formula for calculating an allowance can be considered as a fact) or some property of the software system meaningful in the application domain. FactUnit may be also associated with an arbitrary collection of the KDM entities. A FactUnit then provides a representation of such property inside the KDM model which can be used for further analysis and later exported into a business rule modeling tool in the process known as application business rules mining. The FactUnit class is aligned with SBVR fact type concept. Semantically, a FactUnit represents a “verb concept,” or an “objectified verb concept,” that can be later used as a noun.

Superclass

ConceptualContainer

Semantics

20.5.4 RuleUnit

The RuleUnit class represents an association between multiple Conceptual entities, such as TermUnit or FactUnit. This association can be unary, binary, or n-ary. From the knowledge mining perspective, a FactUnit may correspond to some condition or constraint in the software system (for example, a condition under which an allowance can be increased, or a statement that a certain property should always be satisfied). RuleUnit also may be associated with an arbitrary collection of the KDM entities (these often involve action elements that represent conditions). A RuleUnit then provides a representation of such condition or constraint inside the KDM model that can be used for further analysis and later exported into a business rule modeling tool in the process known as application business rules mining. The RuleUnit class is aligned with SBVR rule concept. Semantically, a RuleUnit uses some base “verb concept” (usually represented as a fact type) and adds to it obligation, necessity, qualifications, quantifications, conditions, etc.

Superclass

ConceptualContainer

Semantics

20.5.5 ConceptualRole

The ConceptualRole class represents a role played by a participant in a conceptual association, such as a FactUnit or a RuleUnit. ConceptualRole elements are owned by some container, a subclass of ConceptualUnit. The ConceptualRole element provides a placeholder for capturing the name of this role as the “name” attribute of the class. Additional annotations of stereotypes can be provided by using the KDM light-weight extension mechanism.

Superclass

AbstractConceptualUnit

Associations

conceptualElement:AbstractConceptualElement[1]	Represents the participant in the association for the given role.

Semantics

20.5.6 BehaviorUnit Class

The BehaviorUnit class represents an arbitrary collection of KDM entities from the Program Elements layer or Platform Resource layer. From the knowledge discovery perspective, such collection may represent some behavior meaningful in the application domain (or simply interesting for the analysis, understanding, and modernization of the software system). A BehaviorUnit then provides a representation of such behavior inside the KDM model that can be used for further analysis. In particular, larger slices of the program logic can be represented as collections of BehaviorUnit elements linked by ConceptualFlow relationships.

The BehaviorUnit class represents a behavior graph with several paths through the application logic. The "implementation" of this graph is provided by the ActionElements connected with Flow relations, from the Program Elements KDM layer. The graph can be as small as a single ActionElement. BehaviorUnit is an "abstraction" of ActionElements since it provides a modeling element for representing a collection of ActionElements that is meaningful from the application domain perspective, and further manipulate with this representation as a first class citizen of the ConceptualModel of KDM.

Superclass

ConceptualContainer

20.5.7 ScenarioUnit Class

ScenarioUnit represents a path (or multiple related paths) through the behavior graph of the application logic. For example, ScenarioUnit corresponds to a trace through the systems, or a "use case." The "implementation" of this graph is provided by the ActionElements connected with Flow relations, from the Program Elements KDM layer. Semantically, the difference between a BehaviorUnit and a ScenarioUnit is that a BehaviorUnit is an abstraction of behavior, while ScenarioUnit is an abstraction of a trace. For example, an interesting formula, or an algorithm can be represented as a BehaviorUnit rather than a ScenarioUnit. On the other hand, an interesting data flow path through the application can be represented as a ScenarioUnit rather than a BehaviorUnit. ScenarioUnit can own an entire collection of BehaviorUnits, connected with ConceptualFlow elements and can thus represent a slice of the original behavior graph in the implementation of the software system. The conditions responsible for navigation between alternative paths within the graph can be represented as RuleUnits.

Superclass

ConceptualContainer

20.6 ConceptualRelations Class Diagram

ConceptualRelations class diagram defines specific conceptual relationship called ConceptualFlow. The classes and associations involved in the ConceptualRelations class diagram are shown in Figure 20.5.

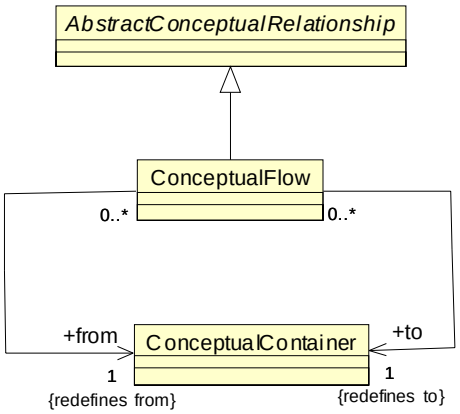

Figure 20.5 - ConceptualRelations Class Diagram

20.6.1 ConceptualFlow Class

The ConceptualFlow class is a KDM relationship defined for the conceptual model. It represents the fact that one behavior may be continued into some other behavior. When multiple ConceptualFlow relations exist for a given conceptual element (according to the KDM relationship encapsulation pattern, these relationships should be owned by the conceptual element and the "from" property of the relationship should be equal to the identity of the element), this means that the behavior represented by that conceptual element may be continued by either of these flows nondeterministically. The follow-up behavior is designated by the conceptual element represented by the "to" property of the ConceptualFlow relationship. When the "to" endpoint of the ConceptualFlow relationship designates a container, this means that any behavior from that container can be the continuation of the initial behavior. When the "from" endpoint of the ConceptualFlow relationship is a container, this means that any behavior element owned by that container can be used as the initial behavior. This relationship provides an abstraction for the Flow relations in the Program Elements layer. ConceptualFlow relation provides a modeling element for representing behavior slices of the application logic that are meaningful from the application domain perspective, and further manipulate with this representation as a first class citizen of the ConceptualModel of KDM.

Superclass

AbstractConceptualRelationship

Associations

from: AbstractConceptualElement[1]	represents the initial behavior
to:AbstractConceptualElement[1]	represents a potential follow-up behavior

Example

```
Form Definition
Program TransactionsApproval  File Name: MM0319.Hfm

…
010 Field1 – Customer ID
011 Field2 – Customer First Name
012 Field3 – Customer Last Name
013 Field4 (list) – Account Number
014 Field5 (list) – Account Type
015 Field6 (list) – Account Balance
…
Program
Program TransactionsApproval  File Name: MM0245.HLa

Program begin
…..
100 // Definitions of variables mapable to the form fields
101 Define Cust_ID(Char 20)
102 Define Cust_FName (Char 25)
103 Define Cust_LName (Char 35)
104 Define Acc_Numb(Char 12)[10]
105 Define Acc_Type(Char 2)[10]
106 Define Acc_Balance(Currency)[10]
107
108 // Definition of other variables
109 Define Bal(Currency)
110 Define Ind(Integer)
111 Define AdjustedBal(Currency)
112 Define ApproveTrans(Boolean)
113 Define Allowance(Currency)
…..

150 // Populating variables entered in the form
151 Field1 -> Cust_ID
152 Field2 -> Cust_FName
153 Field3 -> Cust_LName
154 Field4[1] -> Acc_Numb[0]
155 Field5[1] -> Acc_Type[0]
156 Field6[1] -> Acc_Balance[0]
…
200 // Processing
201 Allowance = $100.00  // The allowance shall be calculated for each customer
202 Ind =1
203 Bal = Acc_Balance[Ind – 1]
204 AdjustedBal = Bal + Allowance
…
240 If(AdjustedBal > $1000.00)
241    Then ApproveTrans = True
```

```
242     Else ApproveTrans = False

…
Program end
```

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:action="http://schema.omg.org/spec/KDM/1.2/action"
       xmlns:code="http://schema.omg.org/spec/KDM/1.2/code"
       xmlns:conceptual="http://schema.omg.org/spec/KDM/1.2/conceptual"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
       xmlns:source="http://schema.omg.org/spec/KDM/1.2/source"
       xmlns:ui="http://schema.omg.org/spec/KDM/1.2/ui" name="Conceptual Example">
  <model xmi:id="id.0" xmi:type="code:CodeModel">
    <codeElement xmi:id="id.1" xmi:type="code:CodeAssembly">
      <codeElement xmi:id="id.2" xmi:type="code:StorableUnit" name="Cust_ID"
             type="id.127" ext="Char 20" size="20">
        <comment xmi:id="id.3" text="// Definitions of variables mapable to the form fields"/>
      </codeElement>
      <codeElement xmi:id="id.4" xmi:type="code:StorableUnit" name="Cust_FName"
             type="id.127" ext="Char 25" size="25"/>
      <codeElement xmi:id="id.5" xmi:type="code:StorableUnit" name="Cust_LName"
             type="id.127" ext="Char 35" size="35"/>
      <codeElement xmi:id="id.6" xmi:type="code:StorableUnit" name="Acc_Numb"
             type="id.7" ext="" size="1">
        <codeElement xmi:id="id.7" xmi:type="code:ArrayType" size="10">
          <itemUnit xmi:id="id.8" name="Acc_Numb[]" type="id.127" ext="Char 12" size="12"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.9" xmi:type="code:StorableUnit" name="Acc_Type"
             type="id.10" ext="" size="1">
        <codeElement xmi:id="id.10" xmi:type="code:ArrayType" size="10">
          <itemUnit xmi:id="id.11" name="Acc_Type[]" type="id.127" ext="Char 2" size="2"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.12" xmi:type="code:StorableUnit" name="Acc_Balance"
                type="id.13" ext="" size="1">
        <codeElement xmi:id="id.13" xmi:type="code:ArrayType" size="10">
          <itemUnit xmi:id="id.14" name="Acc_Balance[]" type="id.128" ext="Currency" size="2"/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id="id.15" xmi:type="code:StorableUnit" name="Bal"
             type="id.128" ext="" size="1" kind="local">
        <comment xmi:id="id.16" text="// Definition of other variables"/>
      </codeElement>
      <codeElement xmi:id="id.17" xmi:type="code:StorableUnit" name="Ind"
             type="id.129" ext="" size="1" kind="local"/>
      <codeElement xmi:id="id.18" xmi:type="code:StorableUnit" name="AdjustedBal"
             type="id.128" ext="" size="1" kind="local"/>
```

```
<codeElement xmi:id="id.19" xmi:type="code:StorableUnit" name="ApprovedTrans"
       type="id.130" ext="" size="1" kind="local"/>
<codeElement xmi:id="id.20" xmi:type="code:StorableUnit" name="Allowance"
       type="id.128" ext="" size="1" kind="local"/>
<codeElement xmi:id="id.21" xmi:type="action:ActionElement" name="i1" kind="Assign">
  <source xmi:id="id.22" language="Hla" snippet="Field1 -> Cust_ID"/>
  <comment xmi:id="id.23" text="// Populating variables entered in the form"/>
  <codeElement xmi:id="id.24" xmi:type="code:StorableUnit" name="Field1"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.25" xmi:type="action:Reads" to="id.24" from="id.21"/>
  <actionRelation xmi:id="id.26" xmi:type="action:Writes" to="id.2" from="id.21"/>
  <actionRelation xmi:id="id.27" xmi:type="action:Flow" to="id.28" from="id.21"/>
</codeElement>
<codeElement xmi:id="id.28" xmi:type="action:ActionElement" name="i2" kind="Assign">
  <source xmi:id="id.29" language="Hla" snippet="Field2 -> Cust_FName"/>
  <codeElement xmi:id="id.30" xmi:type="code:StorableUnit" name="Field2"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.31" xmi:type="action:Reads" to="id.30" from="id.28"/>
  <actionRelation xmi:id="id.32" xmi:type="action:Writes" to="id.4" from="id.28"/>
  <actionRelation xmi:id="id.33" xmi:type="action:Flow" to="id.34" from="id.28"/>
</codeElement>
<codeElement xmi:id="id.34" xmi:type="action:ActionElement" name="i3" kind="Assign">
  <source xmi:id="id.35" language="Hla" snippet="Field3 -> Cust_LName"/>
  <codeElement xmi:id="id.36" xmi:type="code:StorableUnit" name="Field3"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.37" xmi:type="action:Reads" to="id.36" from="id.34"/>
  <actionRelation xmi:id="id.38" xmi:type="action:Writes" to="id.5" from="id.34"/>
  <actionRelation xmi:id="id.39" xmi:type="action:Flow" to="id.40" from="id.34"/>
</codeElement>
<codeElement xmi:id="id.40" xmi:type="action:ActionElement" name="i4" kind="ArrayReplace">
  <source xmi:id="id.41" language="Hla" snippet="Field5[1] -> Acc_Type[0]"/>
  <codeElement xmi:id="id.42" xmi:type="code:Value" name="0" type="id.129"/>
  <codeElement xmi:id="id.43" xmi:type="code:StorableUnit" name="Field4"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.44" xmi:type="action:Reads" to="id.42" from="id.40"/>
  <actionRelation xmi:id="id.45" xmi:type="action:Addresses" to="id.9" from="id.40"/>
  <actionRelation xmi:id="id.46" xmi:type="action:Reads" to="id.43" from="id.40"/>
  <actionRelation xmi:id="id.47" xmi:type="action:Writes" to="id.8" from="id.40"/>
  <actionRelation xmi:id="id.48" xmi:type="action:Flow" to="id.49" from="id.40"/>
</codeElement>
<codeElement xmi:id="id.49" xmi:type="action:ActionElement" name="i5" kind="ArrayReplace">
  <source xmi:id="id.50" language="Hla" snippet="Field4[1] -> Acc_Numb[0]"/>
  <codeElement xmi:id="id.51" xmi:type="code:Value" name="0" type="id.129"/>
  <codeElement xmi:id="id.52" xmi:type="code:StorableUnit" name="Field5"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.53" xmi:type="action:Reads" to="id.51" from="id.49"/>
  <actionRelation xmi:id="id.54" xmi:type="action:Addresses" to="id.6" from="id.49"/>
  <actionRelation xmi:id="id.55" xmi:type="action:Reads" to="id.52" from="id.49"/>
  <actionRelation xmi:id="id.56" xmi:type="action:Writes" to="id.11" from="id.49"/>
  <actionRelation xmi:id="id.57" xmi:type="action:Flow" to="id.58" from="id.49"/>
</codeElement>
```

```
<codeElement xmi:id="id.58" xmi:type="action:ActionElement" name="i6" kind="ArrayReplace">
  <source xmi:id="id.59" language="Hla" snippet="Field6[1] -> Acc_Balance[0]"/>
  <codeElement xmi:id="id.60" xmi:type="code:Value" name="0" type="id.129"/>
  <codeElement xmi:id="id.61" xmi:type="code:StorableUnit" name="Field6"
       type="id.127" kind="register"/>
  <actionRelation xmi:id="id.62" xmi:type="action:Reads" to="id.60" from="id.58"/>
  <actionRelation xmi:id="id.63" xmi:type="action:Addresses" to="id.12" from="id.58"/>
  <actionRelation xmi:id="id.64" xmi:type="action:Reads" to="id.61" from="id.58"/>
  <actionRelation xmi:id="id.65" xmi:type="action:Writes" to="id.14" from="id.58"/>
  <actionRelation xmi:id="id.66" xmi:type="action:Flow" to="id.67" from="id.21"/>
</codeElement>
<codeElement xmi:id="id.67" xmi:type="action:ActionElement" name="p1" kind="Assign">
  <source xmi:id="id.68" language="Hla" snippet="Allowance = $100.00  "/>
  <comment xmi:id="id.69" text="// Processing"/>
  <comment xmi:id="id.70" text="// The allowance shall be calculated for each customer"/>
  <codeElement xmi:id="id.71" xmi:type="code:Value" name="100.00" type="id.128"/>
  <actionRelation xmi:id="id.72" xmi:type="action:Reads" to="id.71" from="id.67"/>
  <actionRelation xmi:id="id.73" xmi:type="action:Writes" to="id.20" from="id.67"/>
  <actionRelation xmi:id="id.74" xmi:type="action:Flow" to="id.75" from="id.67"/>
</codeElement>
<codeElement xmi:id="id.75" xmi:type="action:ActionElement" name="p2" kind="Assign">
  <source xmi:id="id.76" language="Hla" snippet="Ind =1"/>
  <codeElement xmi:id="id.77" xmi:type="code:Value" name="1" type="id.129"/>
  <actionRelation xmi:id="id.78" xmi:type="action:Reads" to="id.77" from="id.75"/>
  <actionRelation xmi:id="id.79" xmi:type="action:Writes" to="id.17" from="id.75"/>
  <actionRelation xmi:id="id.80" xmi:type="action:Flow" to="id.49" from="id.75"/>
</codeElement>
<codeElement xmi:id="id.81" xmi:type="action:ActionElement" name="p3" kind="Compound">
  <source xmi:id="id.82" language="Hla" snippet="Bal = Acc_Balance[Ind - 1]"/>
  <codeElement xmi:id="id.83" xmi:type="code:Value" name="1" type="id.129"/>
  <codeElement xmi:id="id.84" xmi:type="code:StorableUnit" name="t1"
       type="id.129" ext="" kind="register"/>
  <codeElement xmi:id="id.85" xmi:type="action:ActionElement" name="p3.1" kind="Subtract">
    <actionRelation xmi:id="id.86" xmi:type="action:Reads" to="id.17" from="id.81"/>
    <actionRelation xmi:id="id.87" xmi:type="action:Reads" to="id.83" from="id.81"/>
    <actionRelation xmi:id="id.88" xmi:type="action:Writes" to="id.84" from="id.81"/>
    <actionRelation xmi:id="id.89" xmi:type="action:Flow" to="id.90" from="id.85"/>
  </codeElement>
  <codeElement xmi:id="id.90" xmi:type="action:ActionElement" name="p3.2" kind="ArraySelect">
    <actionRelation xmi:id="id.91" xmi:type="action:Addresses" to="id.14" from="id.90"/>
    <actionRelation xmi:id="id.92" xmi:type="action:Reads" to="id.84" from="id.81"/>
    <actionRelation xmi:id="id.93" xmi:type="action:Writes" to="id.15" from="id.81"/>
  </codeElement>
  <actionRelation xmi:id="id.94" xmi:type="action:Flow" to="id.85" from="id.81"/>
  <actionRelation xmi:id="id.95" xmi:type="action:Flow" to="id.96" from="id.81"/>
</codeElement>
<codeElement xmi:id="id.96" xmi:type="action:ActionElement" name="p4" kind="Assign">
  <source xmi:id="id.97" language="Hla" snippet="AdjustedBal = Bal + Allowance"/>
  <actionRelation xmi:id="id.98" xmi:type="action:Reads" to="id.15" from="id.96"/>
  <actionRelation xmi:id="id.99" xmi:type="action:Reads" to="id.20" from="id.96"/>
  <actionRelation xmi:id="id.100" xmi:type="action:Writes" to="id.18" from="id.96"/>
```

```
        <actionRelation xmi:id="id.101" xmi:type="action:Flow" to="id.49" from="id.96"/>
      </codeElement>
      <codeElement xmi:id="id.102" xmi:type="action:ActionElement" name="p5" kind="Assign">
        <source xmi:id="id.103" language="Hla" snippet="If(AdjustedBal > $1000.00)"/>
        <codeElement xmi:id="id.104" xmi:type="code:StorableUnit" name="t2"
                     type="id.130" kind="register"/>
        <codeElement xmi:id="id.105" xmi:type="action:ActionElement" name="p5.1" kind="GreaterThan">
          <codeElement xmi:id="id.106" xmi:type="code:Value" name="1000.00" type="id.128"/>
          <actionRelation xmi:id="id.107" xmi:type="action:Reads" to="id.18" from="id.105"/>
          <actionRelation xmi:id="id.108" xmi:type="action:Reads" to="id.106" from="id.105"/>
          <actionRelation xmi:id="id.109" xmi:type="action:Writes" to="id.104" from="id.105"/>
          <actionRelation xmi:id="id.110" xmi:type="action:Flow" to="id.111" from="id.105"/>
        </codeElement>
        <codeElement xmi:id="id.111" xmi:type="action:ActionElement" name="p5.2" kind="GreaterThan">
          <actionRelation xmi:id="id.112" xmi:type="action:Reads" to="id.104" from="id.111"/>
          <actionRelation xmi:id="id.113" xmi:type="action:TrueFlow" to="id.115" from="id.111"/>
          <actionRelation xmi:id="id.114" xmi:type="action:FalseFlow" to="id.120" from="id.111"/>
        </codeElement>
        <codeElement xmi:id="id.115" xmi:type="action:ActionElement" name="p6" kind="Assign">
          <source xmi:id="id.116" language="Hla" snippet="Then ApproveTrans = True"/>
          <codeElement xmi:id="id.117" xmi:type="code:Value" name="true" type="id.130"/>
          <actionRelation xmi:id="id.118" xmi:type="action:Reads" to="id.117" from="id.115"/>
          <actionRelation xmi:id="id.119" xmi:type="action:Writes" to="id.19" from="id.115"/>
        </codeElement>
        <codeElement xmi:id="id.120" xmi:type="action:ActionElement" name="p7" kind="Assign">
          <source xmi:id="id.121" language="Hla" snippet="Else ApproveTrans = False"/>
          <codeElement xmi:id="id.122" xmi:type="code:Value" name="false" type="id.130"/>
          <actionRelation xmi:id="id.123" xmi:type="action:Reads" to="id.122" from="id.120"/>
          <actionRelation xmi:id="id.124" xmi:type="action:Writes" to="id.19" from="id.120"/>
        </codeElement>
        <actionRelation xmi:id="id.125" xmi:type="action:Flow" to="id.105" from="id.102"/>
      </codeElement>
    </codeElement>
    <codeElement xmi:id="id.126" xmi:type="code:LanguageUnit">
      <codeElement xmi:id="id.127" xmi:type="code:StringType"/>
      <codeElement xmi:id="id.128" xmi:type="code:DecimalType" name="Currency"/>
      <codeElement xmi:id="id.129" xmi:type="code:IntegerType"/>
      <codeElement xmi:id="id.130" xmi:type="code:BooleanType"/>
    </codeElement>
  </model>
  <model xmi:id="id.131" xmi:type="source:InventoryModel">
    <inventoryElement xmi:id="id.132" xmi:type="source:Directory" path="SOURCES\HLanguage">
      <inventoryElement xmi:id="id.133" xmi:type="source:SourceFile" name="mm0245.Hla"/>
      <inventoryElement xmi:id="id.134" xmi:type="source:SourceFile" name="mm0319.Hfm"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.135" xmi:type="source:Directory" path="SOURCES\Hlib"/>
  </model>
  <model xmi:id="id.136" xmi:type="ui:UIModel">
    <UIElement xmi:id="id.137" xmi:type="ui:Screen" name="Customer Information">
      <UIElement xmi:id="id.138" xmi:type="ui:UIField" name="Customer ID">
        <abstraction xmi:id="id.139" name="f1">
```

```
        <actionRelation xmi:id="id.140" xmi:type="action:Writes" to="id.24" from="id.139"/>
      </abstraction>
    </UIElement>
    <UIElement xmi:id="id.141" xmi:type="ui:UIField" name="Customer First Name">
      <abstraction xmi:id="id.142" name="f2">
        <actionRelation xmi:id="id.143" xmi:type="action:Writes" to="id.30" from="id.142"/>
      </abstraction>
    </UIElement>
    <UIElement xmi:id="id.144" xmi:type="ui:UIField" name="Customer Last Name">
      <abstraction xmi:id="id.145" name="f3">
        <actionRelation xmi:id="id.146" xmi:type="action:Writes" to="id.36" from="id.145"/>
      </abstraction>
    </UIElement>
    <UIElement xmi:id="id.147" xmi:type="ui:UIField" name="Account Number">
      <abstraction xmi:id="id.148" name="f4">
        <actionRelation xmi:id="id.149" xmi:type="action:Writes" to="id.43" from="id.148"/>
      </abstraction>
    </UIElement>
    <UIElement xmi:id="id.150" xmi:type="ui:UIField" name="Account Type">
      <abstraction xmi:id="id.151" name="f5">
        <actionRelation xmi:id="id.152" xmi:type="action:Writes" to="id.52" from="id.151"/>
      </abstraction>
    </UIElement>
    <UIElement xmi:id="id.153" xmi:type="ui:UIField" name="Account Balance">
      <abstraction xmi:id="id.154" name="f6">
        <actionRelation xmi:id="id.155" xmi:type="action:Writes" to="id.61" from="id.154"/>
      </abstraction>
    </UIElement>
  </UIElement>
</model>
<model xmi:id="id.156" xmi:type="conceptual:ConceptualModel" name="Customer Information">
  <conceptualElement xmi:id="id.157" xmi:type="conceptual:TermUnit" name="AccountBalance"
                     implementation="id.15 id.12 id.17 id.153"/>
  <conceptualElement xmi:id="id.158" xmi:type="conceptual:TermUnit" name="MaxAdjustedBalance"
                     implementation="id.106"/>
  <conceptualElement xmi:id="id.159" xmi:type="conceptual:TermUnit" name="AllowanceAmount"
                     implementation="id.71"/>
  <conceptualElement xmi:id="id.160" xmi:type="conceptual:TermUnit" name="Allowance"
                     implementation="id.20"/>
  <conceptualElement xmi:id="id.161" xmi:type="conceptual:TermUnit" name="AdjustedBalance"
                     implementation="id.18"/>
  <conceptualElement xmi:id="id.162" xmi:type="conceptual:TermUnit" name="AccountBalanceField"
                     implementation="id.153"/>
  <conceptualElement xmi:id="id.163" xmi:type="conceptual:FactUnit"
                     name="AdjustedBalanceUnderThreshold" implementation="id.105">
    <conceptualRelation xmi:id="id.164" xmi:type="conceptual:ConceptualFlow"
           to="id.178" from="id.163"/>
    <conceptualRelation xmi:id="id.165" xmi:type="conceptual:ConceptualFlow"
           to="id.183" from="id.163"/>
    <conceptualElement xmi:id="id.166" xmi:type="conceptual:ConceptualRole" name="Adjusted Balance"
            conceptualElement="id.161"/>
```

```
  <conceptualElement xmi:id="id.167" xmi:type="conceptual:ConceptualRole" name="Threshold"
           conceptualElement="id.158"/>
</conceptualElement>
<conceptualElement xmi:id="id.168" xmi:type="conceptual:FactUnit" name="AccountBalanceCalculation"
           implementation="id.58 id.75 id.81">
  <conceptualRelation xmi:id="id.169" xmi:type="conceptual:ConceptualFlow"
         to="id.172" from="id.168"/>
  <conceptualElement xmi:id="id.170" xmi:type="conceptual:ConceptualRole" name="Boundary element"
           conceptualElement="id.162"/>
  <conceptualElement xmi:id="id.171" xmi:type="conceptual:ConceptualRole" name="Account"
           conceptualElement="id.157"/>
</conceptualElement>
<conceptualElement xmi:id="id.172" xmi:type="conceptual:FactUnit"
           name="AdjustedBalanceCalculation" implementation="id.67 id.96">
  <conceptualRelation xmi:id="id.173" xmi:type="conceptual:ConceptualFlow"
         to="id.163" from="id.172"/>
  <conceptualElement xmi:id="id.174" xmi:type="conceptual:ConceptualRole" name="Account Balance"
           conceptualElement="id.168"/>
  <conceptualElement xmi:id="id.175" xmi:type="conceptual:ConceptualRole" name="Allowance Amount"
           conceptualElement="id.159"/>
</conceptualElement>
<conceptualElement xmi:id="id.176" xmi:type="conceptual:FactUnit" name="TransactionApproved"
           implementation="id.19"/>
<conceptualElement xmi:id="id.177" xmi:type="conceptual:FactUnit" name="TransactionNotApproved"
           implementation="id.19"/>
<conceptualElement xmi:id="id.178" xmi:type="conceptual:RuleUnit" name="ApproveTransaction"
           implementation="id.105 id.111 id.115">
  <source xmi:id="id.179" language="SBVR"
         snippet="Transaction is approved if adjusted balance is under the threshold"/>
  <conceptualRelation xmi:id="id.180" xmi:type="conceptual:ConceptualFlow"
         to="id.176" from="id.178"/>
  <conceptualElement xmi:id="id.181" xmi:type="conceptual:ConceptualRole" name="Condition"
           conceptualElement="id.163"/>
  <conceptualElement xmi:id="id.182" xmi:type="conceptual:ConceptualRole" name="Consequence"
           conceptualElement="id.176"/>
</conceptualElement>
<conceptualElement xmi:id="id.183" xmi:type="conceptual:RuleUnit" name="TransactionFailedApproval"
           implementation="id.105 id.111 id.120">
  <conceptualRelation xmi:id="id.184" xmi:type="conceptual:ConceptualFlow"
                to="id.177" from="id.183"/>
  <conceptualElement xmi:id="id.185" xmi:type="conceptual:ConceptualRole" name="NOT condition"
           conceptualElement="id.163"/>
  <conceptualElement xmi:id="id.186" xmi:type="conceptual:ConceptualRole" name="consequence"
           conceptualElement="id.177"/>
</conceptualElement>
<conceptualElement xmi:id="id.187" xmi:type="conceptual:ScenarioUnit">
  <conceptualElement xmi:id="id.188" xmi:type="conceptual:BehaviorUnit" name="Calculate Balance"
           implementation="id.58 id.75 id.81">
    <conceptualRelation xmi:id="id.189" xmi:type="conceptual:ConceptualFlow"
         to="id.190" from="id.188"/>
  </conceptualElement>
```

```
      <conceptualElement xmi:id="id.190" xmi:type="conceptual:BehaviorUnit"
             name="Calculate Adjusted Balance" implementation="id.67 id.96">
        <conceptualRelation xmi:id="id.191" xmi:type="conceptual:ConceptualFlow"
             to="id.192" from="id.190"/>
      </conceptualElement>
      <conceptualElement xmi:id="id.192" xmi:type="conceptual:BehaviorUnit" name="Approve Transaction"
              implementation="id.102 id.115 id.120"/>
    </conceptualElement>
    <conceptualElement xmi:id="id.193" xmi:type="conceptual:BehaviorUnit" name="Input"
              implementation="id.21 id.28 id.34 id.40 id.49 id.58">
      <conceptualRelation xmi:id="id.194" xmi:type="conceptual:ConceptualFlow"
             to="id.195" from="id.193"/>
    </conceptualElement>
    <conceptualElement xmi:id="id.195" xmi:type="conceptual:BehaviorUnit" name="Processing"
      implementation="id.67 id.75 id.81 id.85 id.90 id.96 id.102 id.105 id.111 id.115 id.120"/>
  </model>
</kdm:Segment>
```

20.7 ExtendedConceptualElements Class Diagram

The ExtendedConceptualElements class diagram defines two “wildcard” generic elements for the conceptual model as determined by the KDM model pattern: a generic conceptual entity and a generic conceptual relationship.

The classes and associations of the ExtendedConceptualElements diagram are shown in Figure 20.6.

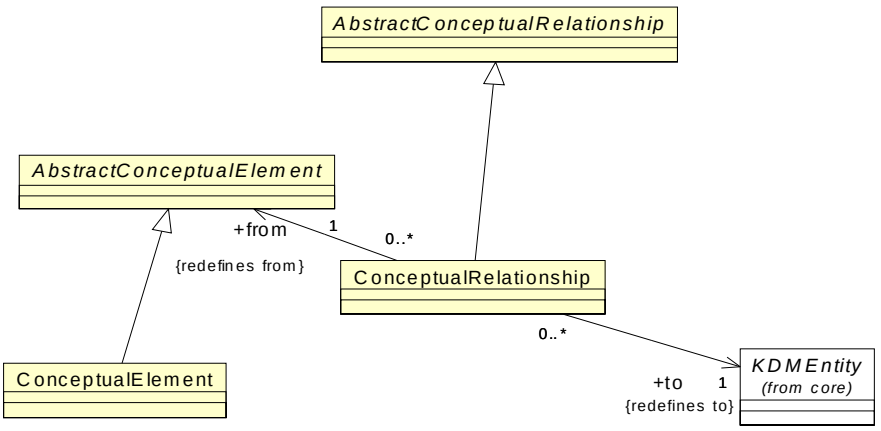

Figure 20.6 - ExtendedConceptualElements Class Diagram

20.7.1 ConceptualElement Class (generic)

The ConceptualElement is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractConceptualElement

Constraints

1. ConceptualElement should have at least one stereotype

Semantics

A conceptual entity with under specified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model entity type of the conceptual model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

20.7.2 ConceptualRelationship Class (generic)

The ConceptualRelationship is a generic meta-model element that can be used to define new "virtual" meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractConceptualRelationship

Associations

from:AbstractConceptualElement[1]	the conceptual element origin of the relationship
to:KDMEntity[1]	the KDMEntity target of the relationship

Constraints

1. ConceptualRelationship should have at least one stereotype.

Semantics

A conceptual relationship with underspecified semantics. It is a concrete class that can be used as the base element of a new "virtual" meta-model relationship type of the conceptual model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

21 Build Package

21.1 Overview

The Build package defines meta-model elements that represent the facts involved in the build process of the given software system (including but not limited to the engineering transformations of the “source code” to “executables”). The Build package also includes the meta-model elements to represent the artifacts that are generated by the build process.

The Build package defines an *architectural viewpoint* for the Build domain.

- **Concerns**:
  - What are the inputs to the build process?
  - What artifacts are generated during the build process?
  - What tools are used to perform build steps?
  - What is the workflow of the build process?
  - Who are the suppliers of the source artifacts?
- **Viewpoint language**

  Build views conform to KDM XMI schema. The *viewpoint language* for the Build *architectural viewpoint* is defined by the Build package. It includes abstract entity AbstractBuildElement, a generic entity BuildResource as well as several concrete entities, such as BuildComponent, BuildStep, BuildProduct, BuildDescription, Library. The viewpoint language for the Build architectural viewpoint also includes several build relationships, which is a subclass of an abstract relationship AbstractBuildRelationship.
- **Analytic methods**
  - Supply chain analysis (what are the artifacts that depend on a given supplier)

  Build Views are used in combination with Inventory views.
- **Construction methods**:
  - Build views that correspond to the KDM Build architectural viewpoint are usually constructed by analyzing build scripts and build configuration files for the given system. This inputs are specific to the build automation framework. The Build extractor tool uses the knowledge of the semantics of the build automation framework to produce one or mode Build views as output.
  - Construction of the Build view is determined by the semantics of the build automation framework, and it based on the mapping from the given build automation framework to KDM; such mapping is specific only to the build automation framework and not to a specific software system.
  - The mapping from a particular build automation framework to KDM may produce additional information (system-specific, or platform-specific, or extractor tool-specific). This information can be attached to KDM elements using stereotypes, attributes, or annotations.

21.2 Organization of the Build Package

The Build package defines meta-model elements that represent entities and relationships related to the build process of an existing software system.

The Build package consists of the following class diagrams.

1. BuildModel
2. BuildInheritances
3. BuildResources
4. BuildRelations
5. ExtendedBuildRelations

The Build package depends on the following packages:

- Core
- kdm
- Source

21.3 BuildModel Class Diagram

The BuildModel class diagram provides basic meta-model elements that represent entities and relationships related to the build process of an existing software system. The class diagram shown in Figure 21.1 captures these classes and their relations.

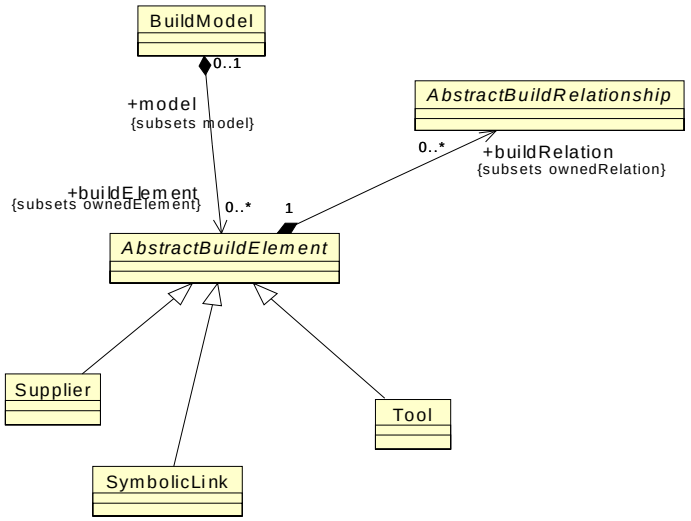

Figure 21.1 - BuildModel Class Diagram

21.3.1 BuildModel Class

The BuildModel encapsulates meta-model constructs needed to model the building of a particular software system.

Superclass

KDMModel

Associations

buildElement:AbstractBuildElement[0..*]	The set of build elements owned by the model.

Semantics

21.3.2 AbstractBuildElement Class (abstract)

The AbstractBuildElement is the abstract base class from which all other build model elements are extended.

Superclass

KDMEntity

Associations

buildRelationship:AbstractBuildRelationship[0..*]	the set of build relations

Semantics

21.3.3 AbstractBuildRelationship Class (abstract)

The AbstractBuildRelationship is the abstract base class.

Superclass

KDMRelationship

Semantics

21.3.4 Supplier Class

The Supplier class models producers of the 3rd party software components as they contribute to the build process.

Superclass

AbstractBuildElement

Semantics

21.3.5 Tool Class

The Tool class represents software tools as they are used in the build process.

Superclass

AbstractBuildElement

Semantics

21.3.6 SymbolicLink Class

The SymbolicLink is used to represent symbolic links.

Superclass

AbstractBuildElement

Semantics

21.4 BuildInheritances Class Diagram

The BuildInheritances class diagram shown in Figure 21.2 depicts how various build classes extend other KDM classes. Each of the classes shown in this diagram inherits properties from classes found in the KDM Core package.

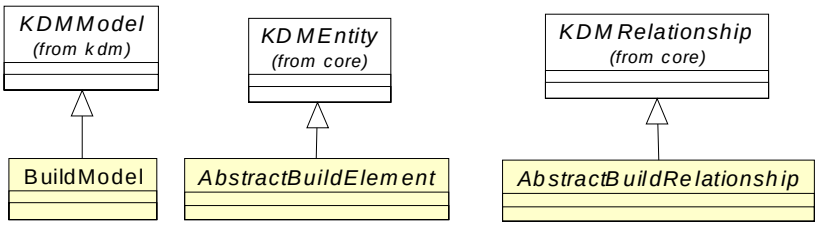

Figure 21.2 - BuildInheritances Class Diagram

21.5 BuildResources Class Diagram

The BuildResources class diagram provides basic meta-model constructs to model various common build resource and their relations to the code model. The class diagram shown in Figure 21.3 captures these classes and their relations.

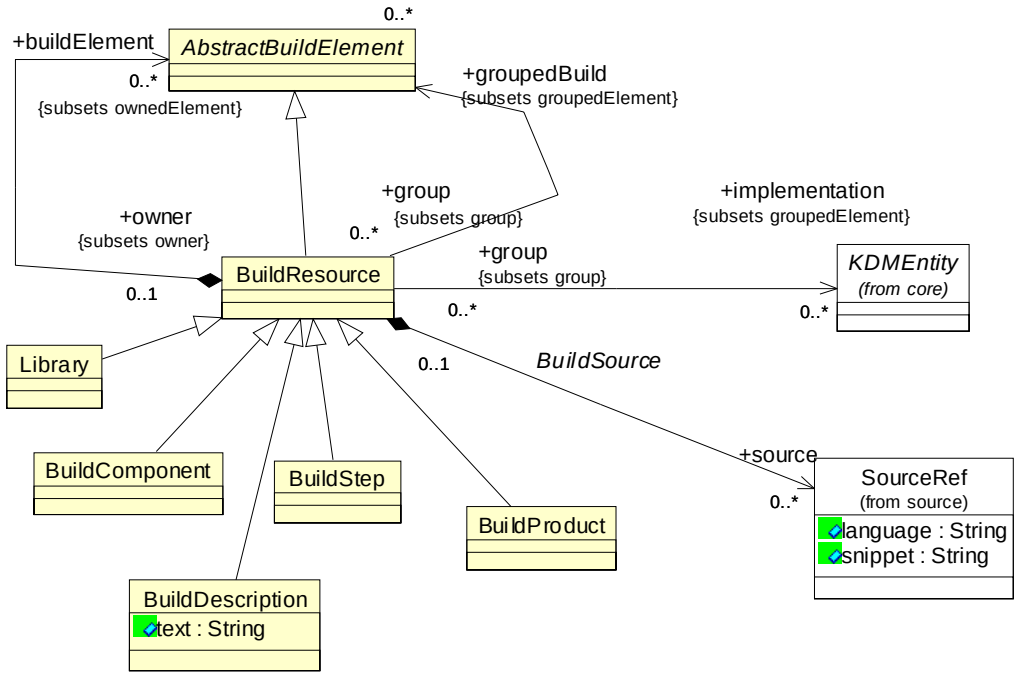

Figure 21.3 - BuildResources Class Diagram

21.5.1 BuildResource Class

BuildResource class is a generic meta-model element that represents a container for build elements. It provides a common superclass for the build elements that can own other build elements. BuildResource is also a group for other KDM entities. Usually, a Build resource such as a Library, a BuildProduct, or a BuildComponent will group together some Inventory elements. Certain BuildResource can also group other build elements.

Superclass

AbstractBuildElement

Associations

buildElement:AbstractBuildElement[0..*]	owned build element
groupedBuild:AbstractBuildElement[0..*]	grouped build elements (KDM group mechanism)
implementation:KDMEntity[0..*]	
source:SourceRef[0..*]	Link to the physical artifact which is represented by the BuildResource element

Constraints

1. BuildResource should either own elements or group elements, but not both.

2. “Implementation” group should not include other Build elements.

3. Build element should not be included in its own groupedBuild group.

Semantics

21.5.2 BuildComponent Class

The BuildComponent class represents binary files that correspond to deployable components, for example executable files.

Superclass

BuildResource

Semantics

21.5.3 BuildDescription Class

The BuildDescription class is used to model objects such as make files or ant scripts, which describe the build process itself.

Superclass

BuildResource

Semantics

21.5.4 BuildStep Class

BuildStep class is the key meta-model element of the Build model. It represents a unit of transformation performed by the build process, during which certain input resources are processed and certain output resources are produced. BuildStep element is the origin of several build relationships. For example, a Build step “consumes” certain input resources, “produces” certain output resources, “is defined” by a certain build description, and may be “supported” by a certain tool.

Superclass

BuildResource

Semantics

21.6 BuildRelations Class Diagram

The BuildRelations class diagram defines the various build related relationships. The class diagram shown in Figure 21.4 captures these classes and their relations.

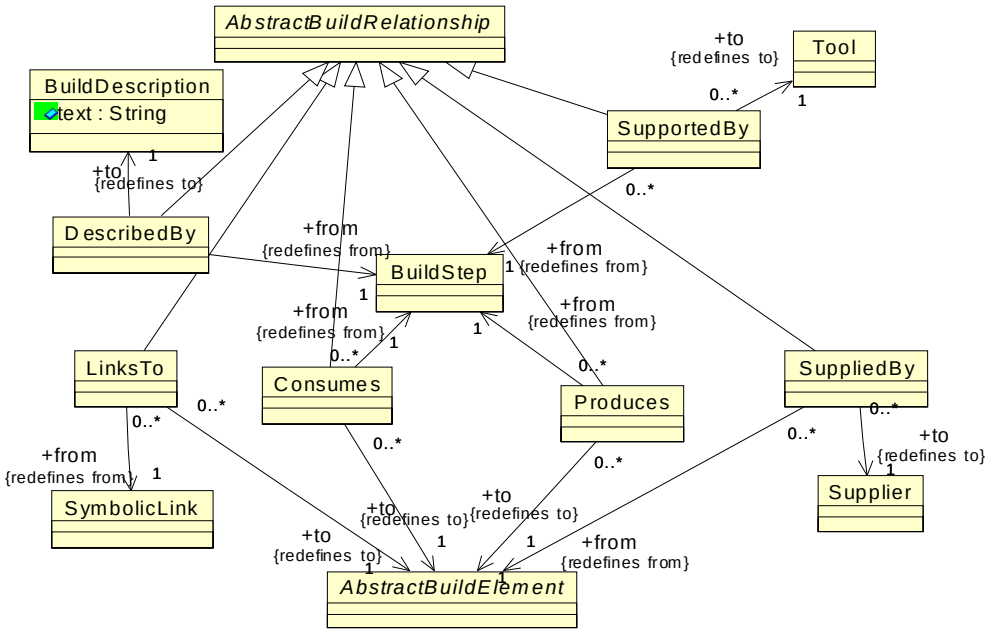

Figure 21.4 - BuildRelations Class Diagram

21.6.1 LinksTo Class

The LinksTo class models the relationship between two linked build resources.

Superclass

AbstractBuildRelationship

Associations

from:SymbolicLink[1]

to:AbstractBuildElement[1]

Semantics

Associations

from:AbstractBuildElement[1]

to:AbstractBuildElement[1]

Semantics

21.6.2 Consumes Class

Consumes class defines association between a certain BuildStep element and certain build elements, called the input build elements. These elements provide the input to the transformation, performed by the build step. For example, the set of source files is an input to the compilation step.

Superclass

AbstractBuildRelationship

Associations

from:BuildStep[1]	the build step
to:AbstractBuildElement[1]	the input build elements for the given step

Semantics

The implementer shall capture the input build elements for a certain build step in the form of “Consumes” relation.

When the target of the “Consumes” relationship owns other build elements, this means that the build step (the origin of the relationship) depends on all elements owned by the container (directly or indirectly).

When the origin of the “Consumes” relationship is a container that owns one or more build steps (directly or indirectly), this means that all build steps consume the elements designated as the target of the “Consumes” relationship.

21.6.3 Produces Class

Produces class defines association between a certain BuildStep element and certain build elements, called the output build elements. These elements are produced as the result of the transformation, performed by the build step. For example, the set of object files can be produced as the result of the compilation step.

Superclass

AbstractBuildRelationship

Associations

from:BuildStep[1]	the build step
to:AbstractBuildElement[1]	the output build elements for the given step

Semantics

The implementer shall capture the output build elements for a certain build step in the form of “Produces” relation.

When the target of the “Produces” relationship owns other build elements, this means that the build step (the origin of the relationship) produces all elements owned by the container (directly or indirectly).

When the origin of the “Produces” relationship is a container that owns one or more build steps (directly or indirectly), this means that the elements designated as the target of the “Produces” relationship are produced in collaboration of all build steps, and no particular build step is the sole producer.

21.6.4 SupportedBy Class

SupportedBy class defines association between a certain BuildStep element and certain Tool element. The Tool element is required to perform the build step. For example, a particular version of a complier is required to perform the compilation step.

Superclass

AbstractBuildRelationship

Associations

from:BuildStep[1]	the build step
to:Tool[1]	The Tool element that represents the tool performing the transformations represented by the given step.

Semantics

The implementer shall capture the required Tool elements for a certain build step in the form of “SupportedBy” relation.

21.6.5 SuppliedBy Class

SuppliedBy class defines association between certain build elements and their points of origin, represented by Supplier element. For example, certain parts of the runtime platform can originate from a software vendor, some libraries can originate from open source.

Superclass

AbstractBuildRelationship

Associations

from:AbstractBuildElement[1]	the build element
to:Supplier[1]	The Supplier element that represents the origin of the build element.

Semantics

The implementer shall capture the origin of build elements in the form of “SuppliedBy” relation.

When the origin of the “SuppliedBy” relationship is a container that owns one or more build elements (directly or indirectly), this means that all elements designated as the source of the “SuppliedBy” relationship are supplied by a particular Supplier element.

21.6.6 DescribedBy Class

DescribedBy class defines association between certain build step and a certain BuildDescription element. These elements are produced as the result of the transformation, performed by the build step. For example, the set of object files can be produced as the result of the compilation step.

Superclass

AbstractBuildRelationship

Associations

from:BuildStep[1]	the build step
to:BuildDescription[1]	The BuildDescription element that describes the transformation represented by the build step.

Semantics

The implementer shall capture the description of a certain build step in the form of “DescribedBy” relation to some BuildDescription element.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<kdm:Segment xmi:version="2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:build="http://schema.omg.org/spec/KDM/1.2/build"
       xmlns:kdm="http://schema.omg.org/spec/KDM/1.2/kdm"
       xmlns:source="http://schema.omg.org/spec/KDM/1.2/source" name="Build Example">
  <model xmi:id="id.0" xmi:type="source:InventoryModel">
    <inventoryElement xmi:id="id.1" xmi:type="source:SourceFile" name="a.c">
      <inventoryRelation xmi:id="id.2" xmi:type="source:DependsOn" to="id.5" from="id.1"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.3" xmi:type="source:SourceFile" name="b.c">
      <inventoryRelation xmi:id="id.4" xmi:type="source:DependsOn" to="id.5" from="id.3"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.5" xmi:type="source:SourceFile" name="ab.h"/>
    <inventoryElement xmi:id="id.6" xmi:type="source:Directory">
      <inventoryElement xmi:id="id.7" xmi:type="source:Image"/>
      <inventoryElement xmi:id="id.8" xmi:type="source:Image"/>
    </inventoryElement>
    <inventoryElement xmi:id="id.9" xmi:type="source:SourceFile" name="makefile"/>
    <inventoryElement xmi:id="id.10" xmi:type="source:ExecutableFile" name="ab.exe"/>
  </model>
  <model xmi:id="id.11" xmi:type="build:BuildModel">
    <buildElement xmi:id="id.12" xmi:type="build:BuildComponent"
           name="sources" implementation="id.1 id.5 id.3"/>
    <buildElement xmi:id="id.13" xmi:type="build:BuildProduct"
           name="ab product" implementation="id.10"/>
    <buildElement xmi:id="id.14" xmi:type="build:BuildStep">
      <buildRelation xmi:id="id.15" xmi:type="build:DescribedBy" to="id.28" from="id.14"/>
```

```
      <buildRelation xmi:id="id.16" xmi:type="build:SupportedBy" to="id.30" from="id.14"/>
      <buildElement xmi:id="id.17" xmi:type="build:BuildStep" name="compile">
        <buildRelation xmi:id="id.18" xmi:type="build:Consumes" to="id.12" from="id.17"/>
        <buildRelation xmi:id="id.19" xmi:type="build:Produces" to="id.25" from="id.17"/>
        <buildRelation xmi:id="id.20" xmi:type="build:SupportedBy" to="id.26" from="id.17"/>
      </buildElement>
      <buildElement xmi:id="id.21" xmi:type="build:BuildStep" name="link">
        <buildRelation xmi:id="id.22" xmi:type="build:Consumes" to="id.25" from="id.21"/>
        <buildRelation xmi:id="id.23" xmi:type="build:Produces" to="id.13" from="id.21"/>
        <buildRelation xmi:id="id.24" xmi:type="build:SupportedBy" to="id.26" from="id.21"/>
      </buildElement>
      <buildElement xmi:id="id.25" xmi:type="build:BuildComponent" name="object files"/>
      <buildElement xmi:id="id.26" xmi:type="build:Tool" name="C compiler">
        <buildRelation xmi:id="id.27" xmi:type="build:SuppliedBy" to="id.32" from="id.26"/>
      </buildElement>
    </buildElement>
    <buildElement xmi:id="id.28" xmi:type="build:BuildDescription" implementation="id.9">
      <source xmi:id="id.29" language="shell" snippet="cc $(SOURCE) -o ab.exe"/>
    </buildElement>
    <buildElement xmi:id="id.30" xmi:type="build:Tool" name="make">
      <buildRelation xmi:id="id.31" xmi:type="build:SuppliedBy" to="id.32" from="id.30"/>
    </buildElement>
    <buildElement xmi:id="id.32" xmi:type="build:Supplier" name="Tools'R'Us corp"/>
  </model>
</kdm:Segment>
```

21.7 ExtendedBuildElements Class Diagram

The ExtendedBuildElements class diagram defines two “wildcard” generic elements for the build model as determined by the KDM model pattern: a generic build entity and a generic build relationship. The classes and associations of the ExtendedBuildElements diagram are shown in Figure 21.5.

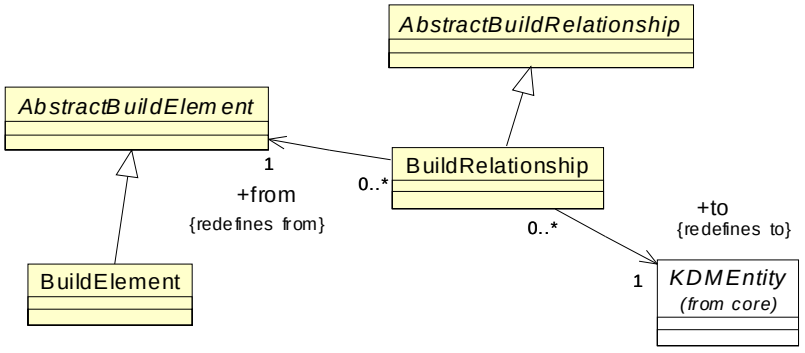

Figure 21.5 - ExtendedBuildElements Class Diagram

21.7.1 BuildElement Class (generic)

The BuildElement is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractBuildElement

Constraints

1. BuildElement should have at least one stereotype.

Semantics

A build entity with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model entity type of the build model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

21.7.2 BuildRelationship Class (generic)

The BuildRelationship is a generic meta-model element that can be used to define new “virtual” meta-model elements through the KDM light-weight extension mechanism.

Superclass

AbstractBuildRelationship

Associations

from:AbstractBuildElement[1]	the build element origin of the relationship
to:KDMEntity[1]	the KDMEntity target of the relationship

Constraints

1. BuildRelationship should have at least one stereotype.

Semantics

A build relationship with under specified semantics. It is a concrete class that can be used as the base element of a new “virtual” meta-model relationship type of the build model. This is one of the KDM *extension points* that can integrate additional language-specific, application-specific, or implementer-specific pieces of knowledge into the standard KDM representation.

Annex A

Semantics of the Micro KDM Action Elements

(normative)

This normative annex defines the semantics of micro KDM action elements. This section assumes understanding of the KDM Datatypes.

Each micro KDM action has the following 5 parts (Action Kind, Inputs, Outputs, Control, and Extras).

- Action Kind - is nature of the operation performed by the micro action. This is represented as a “kind” attribute to the micro action. The action kind may designate certain outgoing relationship as part of the Control. For example, the “call” micro action designated the Calls outgoing relationship as part of Control.
- Outputs - represented by the owned outgoing Writes relationship, which usually represents the result of the micro action. This part is optional.
- Inputs - Ordered outgoing Reads and/or Addresses relationships that are owned by the action element, the order of the relationships represent the order of the arguments for a micro action.
- Control part - owned outgoing control flow relationships for the action.
- Extras part - owned relationships other than Reads, Writes and not designated as part of Control by the action Kind. For example, these can be interface compliance relation “CompliesTo” or any extended action relationships.

A.1 Comparison Actions

Inputs: Two Reads relationships to DataElements representing values of the same datatype (except for Boolean NOT, which has a single Reads relationship).

Outputs: Optional writes to a DataElement of a Boolean type (no Writes corresponds to an expression statement, where the result of the operation is ignored; otherwise, the result should be stored into a DataElement, which can be permanent. For example a StorableUnit with a kind other than “register,” a MemberUnit, an ItemUnit, or a ParameterUnit; or temporary, a StorableUnit with a “register” kind).

Control: Optional single flow - unconditional transfer of control to the next micro action (for example, as part of complex expressions; no Flow corresponds to a terminal action).

Table A.1 - Comparison Actions

Micro action	Semantics
Equals	Polymorphic equals for two values of the same datatype, see ISO Equals operation for the corresponding datatype.
NotEqual	Polymorphic “not equal” for two values of the same datatype: not (A==B).
LessThanOrEqual	Polymorphic “less than or equal” for two values of the same ordered datatype; see ISO InOrder operation for the corresponding datatype.
LessThan	Polymorphic “less than” for two values of the same ordered datatype: A<=B and not A==B.
GreaterThan	Polymorphic “greater than or equal” for two values of the same ordered datatype: not A<=B.
GreaterThanOrEqual	Polymorphic “less than or equal” for two values of the same ordered datatype: not A<=B or A==B.
Not	Boolean NOT, see ISO Boolean NOT operation.
And	Boolean AND, see ISO Boolean AND operation
Or	Boolean OR, see ISO Boolean OR operation
Xor	Boolean XOR: (A and not B) or (not A and B)

A.2 Actions Related to the Primitive Numerical Datatypes

Inputs: Two ordered Reads relationships to DataElements representing values of the same datatype (except for neg, succ, incr, decr unary operations, which have a single Reads relationship).

Outputs: Optional single Writes to a DataElement of a type corresponding to the definition of the operation (can be temporary register or a variable; no Writes corresponds to an expression statement, where the result of the operation is ignored).

Control: Optional single flow - unconditional transfer of control to the next micro action.

Table A.2 - Numerical actions

Micro action	Semantics
Add	Polymorphic add operation for two values of the same numeric datatype, see ISO Add operation for the corresponding datatype.
Multiply	Polymorphic multiply operation for two values of the same numeric datatype; see ISO Add operation for the corresponding datatype.
Negate	Polymorphic unary negate operation for two values of the same numeric datatype; see ISO Negate operation for the corresponding datatype.
Subtract	Polymorphic subtract operation for two values of the same numeric datatype; A+ neg B.
Divide	Polymorphic divide operation for two values of the same numeric datatype.
Remainder	Polymorphic remainder operation for two values of the same IntegerType datatype.
Successor	Single Reads; Successor for ordinal or enumerated types, see ISO Successor operation.

A.3 Actions Related to Bitwise Operations on Primitive Datatypes

Inputs: Two Reads relationships to DataElements representing values of the same datatype (except for neg, succ, incr, decr unary operations, which have a single Reads relationship).

Outputs: Optional single Writes to a DataElement of the same type as the first StorableElement (can be a temporary register or a variable).

Control: Optional single Flow - unconditional transfer of control.

Table A.3 - Bitwise actions

Micro action	Semantics	Inputs
BitAnd	Bitwise AND on two integers or bitstrings or octetstrings	Two Reads relationships to DataElements representing values of the same datatype
BitOr	Bitwise OR on two integers or bitstrings or octetstrings	Two Reads relationships to DataElements representing values of the same datatype
BitNot	Bitwise NOT on integer or bitstring or octetstring	Two Reads relationships to DataElements representing values of the same datatype
BitXor	Bitwise XOR on two integers or bitstrings or octetstrings	Two Reads relationships to DataElements representing values of the same datatype
LeftShift	Arithmetic bitwise shift left on integer or bitstring or octetsting	First Reads relationship to a DataElement representing an integer, bitstring, or octetstring. Second Reads relationship to an integer or ordinal representing the number of bits to shift.
RightShift	Arithmetic bitwise shift right on integer or bitstring or octetstring	First Reads relationship to a DataElement representing an integer, bitstring, or octetstring. Second Reads relationship to an integer or ordinal representing the number of bits to shift.
BitRightShift	Logical bitwise shift right on integer or bitstring or octetstring	First Reads relationship to a DataElement representing an integer, bitstring, or octetstring. Second Reads relationship to an integer or ordinal representing the number of bits to shift.

A.4 Control Actions

Table A.4 - Control actions

Micro action	Description	Inputs	Outputs	Control
Assign	Assignment (copy)	Single Reads relationship to a DataElement representing the value	Writes relationship represents the DataElement (except for a ValueElement) to which the value of the input DataElement is assigned	Optional single flow to the next micro action
Condition	Condition	Single Reads relationship to a DataElement representing the Boolean value	none	TrueFlow & FalseFlow - conditional transfer of control
Call	Static call	Zero or more Reads relationships to DataElements, that represent input actual parameters; ordered; Value of each actual parameter is assigned to the corresponding formal parameter of the ControlElement. Correspondence is established according to the Pos attribute of the formal parameter in the signature of the ControlElement. A sequence of values is assigned to the variable argument.	Optional Writes to the DataElement that represents the return value	Calls relationship to the ControlElement represents the flow of control to the ControlElement and the return back; Subsequently an optional single flow to the next micro action is performed.
MethodCall	Method call	Invokes relationship to the DataElement that represents the instance; Zero or more Reads relationships to DataElements, that represent input actual parameters; ordered;	Same as Call	Calls relationship to the MethodUnit represents the flow of control to the Method and the return back; Subsequently an optional single flow to the next micro action is performed.

Table A.4 - Control actions

Micro action	Description	Inputs	Outputs	Control
PtrCall	Call via pointer	Addresses relationship to the DataElement that represents the pointer; Zero or more Reads relationships to DataElements, that represent input actual parameters; ordered.	Same as Call	This represents a dynamic call to one of the possible targets of the pointer (corresponding to the current value of the pointer). The Signature of the possible targets is represented as the type attribute of the DataElement; subsequently an optional single flow to the next micro action is performed.
VirtualCall	Virtual method call	Invokes relationship to the DataElement that represents the instance; Zero or more Reads relationships to DataElements, that represent input actual parameters; ordered.	Same as Call	Calls relationship to the MethodUnit represents the superclass of the method that will be determined dynamically. This represents the flow of control to the Method and the return back; Subsequently an optional single flow to the next micro action is performed.
Return	return	Single Reads represents the DataElement that contains the return value	none	Control is returned back to one of the ControlElements that has performed the call.
This	pointer to the current instance of the object	none	Writes to a DataElement	Single flow to the next micro action
Nop	dummy	none	none	Optional single flow to the next micro action
Goto	Unconditional transfer of control	none	none	Single flow to the next micro action
Label	represents a label; the name of the action is the label	none	none	Single flow to the next micro action

Table A.4 - Control actions

Micro action	Description	Inputs	Outputs	Control
Throw	Raising exception	none	none	Throws relationship to the DataElement that represents the “exception object.” Optional ExceptionFlow relationship to a CatchUnit that processes the exception.
Incr	Variable post increment operation;	Single Addresses relationship represents the DataElement whose value is incremented;	Optional Writes relationship to another DataUnit to which the previous value of the incremented variable is assigned	Optional single flow to the next micro action
Decr	Variable post decrement operation;	Single Addresses relationship represents the DataElement whose value is decremented	Optional Writes relationship to another DataUnit to which the previous value of the incremented variable is assigned	Optional single flow to the next micro action
Switch	Branching based on the value of a StorableElement	Single Reads to the DataElement that represents the selector value	none	One or more GuardedFlow relations to a second micro action with a single Reads relationship that represents the guard value. A single FalseFlow represents the default branch. This construct represents selection of a single branch for which the value of the selector is equal to the value of the guard or the default branch.

Table A.4 - Control actions

Micro action	Description	Inputs	Outputs	Control
Guard	Represents start of the branch of a complex condition	Single Reads relation to a DataElement representing the guard value	none	Single flow unconditional control flow to the first action of the branch.
Compound	Compound action	none	none	Single Flow - the entry flow to the first internal action element.
Init	BlockUnit that contains initialization action elements	none	none	EntryFlow unconditional control flow to the first internal action.

A.5 Actions Related to Access to Datatypes

Inputs: see table

Outputs: see table

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.5 - Access actions

Micro action	Description	Inputs	Outputs
FieldSelect	Access to a particular ItemUnit of a RecordType	Single Addresses relationship to a DataElement (of a RecordType); Single Reads relationship to an ItemUnit representing the field being accessed.	Optional Writes relationship represents the DataElement (except for a ValueElement) to which the value of the field is assigned.
FieldReplace	Modification of a particular field of a RecordType	Single Addresses relationship to a DataElement (of a RecordType); Single Reads to a DataElement representing the new value.	Writes relationship to an ItemUnit representing the field being modified.
ChoiceSelect	Access to a particular ItemUnit of a ChoiceType	Single Addresses relationship to a DataElement (of a ChoiceType); Single Reads relationship to an ItemUnit representing the field type being accessed.	Optional Writes relationship represents the DataElement (except for a ValueElement) to which the value of the field is assigned.
ChoiceReplace	Modification of a particular field of a ChoiceType	Single Addresses relationship to a DataElement (of a ChoiceType); Single Reads to a DataElement representing the new value.	Writes relationship to an ItemUnit representing the field being modified.

Table A.5 - Access actions

Micro action	Description	Inputs	Outputs
Ptr	Access to a pointer to a StorableElement	Single Addresses relationship to a DataElement	Optional Writes relationship to the StorableElement which will hold the new value.
PtrSelect	Access to a value via pointer	Single Addresses relationship to a DataElement (of an PointerType); Single Reads relationship to an ItemUnit of that PointerType representing the ItemUnit being accessed.	Optional Writes relationship to the ItemUnit of that PointerType.
PtrReplace	Modification of an ItemUnit of a PointerType	Single Addresses relationship to a DataElement (of an PointerType); Last Reads to a DataElement representing the new value.	Writes relationship to the ItemUnit of that PointerType.
ArraySelect	Access to a particular ItemUnit of an ArrayType	Single Addresses relationship to a DataElement (of an ArrayType); Reads relationship to an ItemUnit representing the ItemUnit being accessed; Last Reads represents the Index.	Optinal Writes relationship represents the DataElement (except for a ValueElement) to which the value of the ItemUnit is assigned.
ArrayReplace	Modification of a particular ItemUnit of an ArrayType	Single Addresses relationship to a DataElement (of an ArrayType); Reads that represent the Index; Last Reads to a DataElement representing the new value.	Writes relationship to an ItemUnit representing the ItemUnit being modified.
MemberSelect	Access to a particular MemberUnit of a ClassType	Invokes relationship to the DataElement that represents the instance. Single Reads relationship to a MemberUnit representing the member being accessed.	Optional Writes relationship represents the DataElement (except for a ValueElement) to which the value of the field is assigned.

Table A.5 - Access actions

Micro action	Description	Inputs	Outputs
MemberReplace	Modification of a particular member of a ClassType	Single Invokes relationship to a DataElement (of a ClassType) that represents the instance of the object being accessed. Single Reads to a DataElement representing the new value.	Writes relationship to a MemberUnit representing the member being modified.
New	Creation of a new dynamic instance of a datatype; this has to be done separately if required; this micro action does not invoke the constructor of the new object; this has to be done separately	Creates relationship to the Datatype being created.	Writes relationship represents the DataElement (except for a ValueElement) to which the reference to the new dynamic element is assigned.
NewArray	Creation of a new dynamic instance of an ArrayType datatype	Creates relationship to the Datatype being created; Reads relation to the DataElement that represents the length of the new array.	Writes relationship represents the DataElement (except for a ValueElement) to which the reference to the new dynamic element is assigned.

A.6 Actions Related to Type Conversions

Inputs: see table

Outputs: see table

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.6 - Type conversion actions

Micro action	Description	Inputs	Outputs
Sizeof	Determines the length of a DataElement (based on the datatype) or the length of a Datatype	Reads represents the DataElement; or UsesType to the Datatype	Optional writes to a DataElement.
Instanceof	Performs dynamic type check if the data element is of a certain datatype	Reads represents the DataElement; UsesType relation represents the datatype	Optional Writes to a DataElement of a Boolean type.
DynCast	Performs a dynamic cast of a DataElement to a certain Datatype	Reads represents the DataElement; UsesType relation represents the datatype	Optional Writes to a DataElement.
TypeCast	Performs a static type conversion of a DataElement to a certain Datatype	Reads represents the DataElement; UsesType relation represents the datatype	Optional writes to a DataElement.

A.7 Actions Related to StringType Operations

Inputs: see table

Outputs: optional Writes to a DataElement (no Writes corresponds to an expression statement, where the result of the operation is ignored; otherwise, the result should be stored into a DataElement, which can be permanent. For example a StorableUnit with a kind other than “register,” a MemberUnit, an ItemUnit, or a ParameterUnit; or temporary, a StorableUnit with a “register” kind).

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.7 - StringType actions

Micro action	Description	Inputs
IsEmpty	True is the string x is empty	First Reads represents x;
Head	Produces the value of the first element in the string x	First Reads represents x;
Tail	Produces sequence that results from deleting the first element in the string x	First Reads represents x;
Empty	Produces and empty string	UsesType to the required type
Append	Produces the sequence that is formed by adding a single value y to the end of the string x	First Reads represents x; Second represents y

Note:"==" operation on ISO strings is defined as full comparison, this does not work in Java, which has shallow comparison of object references.

A.8 Actions Related to SetType Operations

Inputs: see table

Outputs: optional Writes to a DataElement (no Writes corresponds to an expression statement, where the result of the operation is ignored; otherwise, the result should be stored into a DataElement, which can be permanent. For example a StorableUnit with a kind other than “register,” a MemberUnit, an ItemUnit, or a ParameterUnit; or temporary, a StorableUnit with a “register” kind).

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.8 - SetType actions

Micro action	Description	Inputs
IsIn	True is the value x is a member of the set y, else false	First Reads represents x; Second represents y
Subset	True if every member of x is a member of y	First Reads represents x; Second represents y
Difference	Produces the set that consists of the values that are in x and not in y	First Reads represents x; Second represents y
Union	Produces the set that consists of the values that are either in x or in y	First Reads represents x; Second represents y
Intersection	Produces the set that consists of the values that are both in x and in y	First Reads represents x; Second represents y
Select	Produces a value of the base type that is in the set x	First Reads represents x;
IsEmpty	True is the set x is empty	First Reads represents x;
Empty	Produces and empty set	UsesType to the required type

A.9 Actions Related to SequenceType Operations

Inputs: see table

Outputs: optional Writes to a DataElement (no Writes corresponds to an expression statement, where the result of the operation is ignored; otherwise, the result should be stored into a DataElement, which can be permanent. For example a StorableUnit with a kind other than “register,” a MemberUnit, an ItemUnit, or a ParameterUnit; or temporary, a StorableUnit with a “register” kind).

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.9 - SequenceType actions

Micro action	Description	Inputs
IsEmpty	True is the sequence x is empty	First Reads represents x;
Head	Produces the value of the first element in the sequence x	First Reads represents x;
Tail	Produces sequence that results from deleting the first element in the sequence x	First Reads represents x;
Empty	Produces and empty sequence	UsesType to the required type
Append	Produces the sequence that is formed by adding a single value y to the end of the sequence x	First Reads represents x; Second represents y

A.10 Actions Related to BagType Operations

Inputs: see table

Outputs: optional Writes to a DataElement (no Writes corresponds to an expression statement, where the result of the operation is ignored; otherwise, the result should be stored into a DataElement, which can be permanent. For example a StorableUnit with a kind other than “register,” a MemberUnit, an ItemUnit, or a ParameterUnit; or temporary, a StorableUnit with a “register” kind).

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Table A.10 - BagType actions

Micro action	Description	Inputs
IsEmpty	True is the bag x is empty	First Reads represents x;
Select	Produces a value of the base type that is in the bag x	First Reads represents x;
Delete	Produces the bag that is formed by deleting one instance of value y from the bag x if any	First Reads represents x; Second represents y
Empty	Produces and empty bag	UsesType to the required type
Insert	Produces the bag that is formed by adding one instance of value y from the bag x	First Reads represents x; Second represents y
Serialize	Produces the sequence in which each element is repeated as many time as it occurs in the bag x	First Reads represents x;

A.11 Actions Related to Resources

Resource micro-actions represent specific statements that are determined by some programming languages and which manipulate resources provided by the operating environment. Such statements are alternative to using system calls. Kinds in Table A.11 represent such statements as micro KDM ActionElements. Precise semantics of representing the operating environment is described in Part 3 Runtime Resource Layer. In particular, a combination of resource actions, resource relationships and resource events can be used, where the resource micro-action is part of a Code Model, while other elements can be added in various models of the Resource Layer (Platform, Data, Event or UI).

Inputs: Zero or more Reads relationships to DataElements; represent input data which is sent to the resource; ordered.

Outputs: Zero or more Writes relationships to DataElements; represents output data which is received from the resource.

Control: optional single Flow to the next micro action (no Flow means a terminal action element).

Extras: optional resource-specific relationships.

Table A.11 - Resource actions

Micro action	Description
Platform	ActionElement represents a statement that manipulates a Platform Resource.
Data	ActionElement represents a statement that manipulates a Data Resource.
Event	ActionElement represents a statement that manipulates an Event Resource.
UI	ActionElement represents a statement that manipulates a UI Resource.

INDEX

D

E

U

V

W

X