
**Information technology — Object
Management Group Architecture-Driven
Modernization (ADM) — Knowledge
Discovery Meta-Model (KDM)**

*Technologies de l'information — Modernisation conduite par
l'architecture (ADM) de l'OMG — Métamodèle de découverte de
connaissances (KDM)*





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2012

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Table of Contents

Foreword	xv
Introduction	xvi
1 Scope	1
2 Conformance and Compliance	1
2.1 KDM Domains	2
2.2 Compliance Levels	2
2.3 Meaning and Types of Compliance	3
3 Normative References	6
4 Terms and Definitions	6
5 Symbols	8
6 Additional Information	9
6.1 Changes to Other OMG Specifications	9
6.2 How to Read this International Standard	9
7 Overview	11
8 KDM	15
8.1 Overview	15
8.2 Organization of the KDM Packages	16
<i>Subpart I - Infrastructure Layer</i>	19
9 Core Package	21
9.1 Overview	21
9.2 Organization of the Core Package	21
9.3 CoreEntities Class Diagram	21
9.3.1 Element Class (abstract)	22
9.3.2 ModelElement Class (abstract)	22
9.3.3 KDMEntity Class (abstract)	23
9.4 CoreRelations Class Diagram	24
9.4.1 KDMRelationship Class (abstract)	24
9.4.2 KDMEntity (additional properties)	25

9.5	AggregatedRelations Class Diagram	26
9.5.1	AggregatedRelationship Class	26
9.5.2	KDMEntity (additional properties)	29
9.6	Datatypes Class Diagram	29
9.6.1	Boolean Type (datatype)	29
9.6.2	String Type (datatype)	29
9.6.3	Integer Type (datatype)	30
10	The Package named “kdm”	31
10.1	Overview	31
10.2	Organization of the KDM Framework	31
10.3	Framework Class Diagram	32
10.3.1	KDMFramework Class (abstract)	33
10.3.2	KDMModel Class (abstract)	33
10.3.3	KDMEntity (additional properties)	34
10.3.4	Segment Class	34
10.4	Audit Class Diagram	35
10.4.1	Audit Class	36
10.4.2	KDMFramework (additional properties)	37
10.5	Extensions Class Diagram	37
10.5.1	Stereotype Class	38
10.5.2	TagDefinition Class	40
10.5.3	ExtensionFamily Class	41
10.5.4	ModelElement (additional properties)	42
10.6	ExtendedValues Class Diagram	43
10.6.1	ExtendedValue Class (abstract)	43
10.6.2	TaggedValue Class	44
10.6.3	TaggedRef Class	44
10.7	Annotations Class Diagram	45
10.7.1	Attribute Class	46
10.7.2	Annotation Class	47
10.7.3	Element (additional properties)	47
11	Source Package	49
11.1	Overview	49
11.2	Organization of the Source Package	50
11.3	InventoryModel Class Diagram	51
11.3.1	InventoryModel Class	51
11.3.2	AbstractInventoryElement Class (abstract)	52
11.3.3	AbstractInventoryRelationship Class (abstract)	52
11.3.4	InventoryItem Class (generic)	53

11.3.5 SourceFile Class	53
11.3.6 Image Class	54
11.3.7 Configuration Class	54
11.3.8 ResourceDescription Class	54
11.3.9 BinaryFile Class	54
11.3.10 ExecutableFile Class	55
11.3.11 InventoryContainer Class (generic)	55
11.3.12 Directory Class	55
11.3.13 Project Class	56
11.4 InventoryInheritances Class Diagram	56
11.5 InventoryRelations Class Diagram	57
11.5.1 DependsOn Class	57
11.6 SourceRef Class Diagram	58
11.6.1 SourceRef Class	58
11.6.2 SourceRegion Class	60
11.7 ExtendedInventoryElements Class Diagram	61
11.7.1 InventoryElement Class (generic)	61
11.7.2 InventoryRelationship Class (generic)	61
<i>Subpart II - Program Elements Layer</i>	63
12 Code Package	67
12.1 Overview	67
12.2 Organization of the Code Package	67
12.3 CodeModel Class Diagram	68
12.3.1 CodeModel Class	69
12.3.2 AbstractCodeElement Class (abstract)	69
12.3.3 AbstractCodeRelationship Class (abstract)	70
12.3.4 CodeItem Class (abstract)	70
12.3.5 ComputationalObject Class (generic)	70
12.3.6 Datatype Class (generic)	71
12.4 CodeInheritances Class Diagram	71
12.5 Modules Class Diagram	72
12.5.1 Module Class (generic)	72
12.5.2 CompilationUnit Class	73
12.5.3 SharedUnit Class	73
12.5.4 LanguageUnit Class	74
12.5.5 CodeAssembly Class	74
12.5.6 Package Class	74
12.6 ControlElements Class Diagram	75
12.6.1 ControlElement Class (generic)	75
12.6.2 CallableUnit Class	76
12.6.3 CallableKind Data Type (enumerated)	77

12.6.4 MethodUnit Class	77
12.6.5 MethodKind data type (enumeration)	77
12.7 DataElements Class Diagram	79
12.7.1 DataElement Class (generic)	80
12.7.2 StorableUnit Class	80
12.7.3 StorableKind data type (enumeration)	81
12.7.4 ExportKind data type (enumeration)	81
12.7.5 ItemUnit Class	82
12.7.6 IndexUnit Class	82
12.7.7 MemberUnit Class	82
12.7.8 ParameterUnit Class	83
12.8 ValueElements Class Diagram	83
12.8.1 ValueElement Class (generic)	84
12.8.2 Value Class	84
12.8.3 ValueList Class	85
12.9 PrimitiveTypes Class Diagram	86
12.9.1 PrimitiveType Class (generic)	86
12.9.2 BooleanType Class	87
12.9.3 CharType Class	87
12.9.4 OrdinalType Class	87
12.9.5 DateType Class	87
12.9.6 TimeType Class	88
12.9.7 IntegerType Class	88
12.9.8 DecimalType Class	88
12.9.9 ScaledType Class	89
12.9.10 FloatType Class	89
12.9.11 VoidType Class	89
12.9.12 StringType Class	89
12.9.13 BitType Class	90
12.9.14 BitstringType Class	90
12.9.15 OctetType Class	90
12.9.16 OctetstringType Class	90
12.10 EnumeratedTypes Class Diagram	91
12.10.1 EnumeratedType Class	91
12.11 CompositeTypes Class Diagram	92
12.11.1 CompositeType Class (generic)	92
12.11.2 ChoiceType Class	93
12.11.3 RecordType Class	93
12.12 DerivedTypes Class Diagram	95
12.12.1 DerivedType Class (generic)	95
12.12.2 ArrayType Class	96
12.12.3 PointerType Class	96
12.12.4 RangeType Class	97
12.12.5 BagType Class	98
12.12.6 SetType Class	98

12.12.7 SequenceType Class	99
12.13 Signature Class Diagram	99
12.13.1 Signature Class	99
12.13.2 ParameterKind Enumeration Datatype	100
12.14 DefinedTypes Class Diagram	100
12.14.1 DefinedType Class (abstract)	101
12.14.2 TypeUnit Class	101
12.14.3 SynonymUnit Class	102
12.15 ClassTypes Class Diagram	102
12.16 ClassUnit Class	102
12.16.1 InterfaceUnit Class	103
12.17 Templates Class Diagram	103
12.17.1 TemplateUnit Class	104
12.17.2 TemplateParameter Class	104
12.17.3 TemplateType Class	105
12.18 TemplateRelations Class Diagram	105
12.18.1 InstanceOf Class	106
12.18.2 ParameterTo Class	106
12.19 InterfaceRelations Class Diagram	110
12.19.1 Implements Class	111
12.19.2 ImplementationOf Class	112
12.20 TypeRelations Class Diagram	115
12.20.1 HasType Class	116
12.20.2 HasValue Class	116
12.21 ClassRelations Class Diagram	121
12.21.1 Extends Class	121
12.22 Preprocessor Class Diagram	123
12.22.1 PreprocessorDirective Class (generic)	123
12.22.2 MacroUnit Class	125
12.22.3 MacroKind data type (enumeration)	125
12.22.4 MacroDirective Class	125
12.22.5 IncludeDirective Class	126
12.22.6 Conditional Directive Class	126
12.23 PreprocessorRelations Class Diagram	126
12.23.1 Expands Class	127
12.23.2 GeneratedFrom Class	128
12.23.3 Includes Class	130
12.23.4 VariantTo Class	131
12.23.5 Redefines Class	133
12.24 Comments Class Diagram	135

12.24.1 CommentUnit Class	135
12.24.2 AbstractCodeElement Class (additional properties)	136
12.25 Visibility Class Diagram	136
12.25.1 Namespace Class	136
12.26 VisibilityRelations Class Diagram	137
12.26.1 VisibleIn Class	137
12.26.2 Imports Class	138
12.27 ExtendedCodeElements Class Diagram	139
12.27.1 CodeElement Class (generic)	140
12.27.2 CodeRelationship Class (generic)	140
13 Action Package	141
13.1 Overview	141
13.2 Organization of the Action Package	141
13.3 ActionElements Class Diagram	141
13.3.1 ActionElement Class	142
13.3.2 AbstractActionRelationship Class (abstract)	143
13.3.3 BlockUnit Class	143
13.3.4 AbstractCodeElement (additional properties)	144
13.4 ActionInheritances Class Diagram	144
13.5 ActionFlow Class Diagram	145
13.5.1 ControlFlow Class (generic)	145
13.5.2 EntryFlow Class	146
13.5.3 Flow Class	147
13.5.4 TrueFlow Class	147
13.5.5 FalseFlow Class	148
13.5.6 GuardedFlow Class	148
13.6 CallableRelations Class Diagram	149
13.6.1 Calls Class	150
13.6.2 Dispatches Class	151
13.7 DataRelations Class Diagram	152
13.7.1 Reads Class	153
13.7.2 Writes Class	154
13.7.3 Addresses Class	154
13.7.4 Creates Class	154
13.8 ExceptionBlocks Class Diagram	155
13.8.1 ExceptionUnit Class	155
13.8.2 TryUnit Class	156
13.8.3 CatchUnit Class	156
13.8.4 FinallyUnit Class	157
13.9 ExceptionFlow Class Diagram	159

13.9.1 ExitFlow Class	160
13.9.2 ExceptionFlow Class	161
13.10 ExceptionRelations Class Diagram	161
13.10.1 Throws Class	162
13.11 InterfaceRelations Class Diagram	162
13.11.1 CompliesTo Class	163
13.12 UsesRelations Class Diagram	164
13.12.1 UsesType Class	164
13.13 ExtendedActionElements Class Diagram	164
13.13.1 ActionRelationship Class (generic)	165
14 Micro KDM	167
14.1 Overview	167
<i>Subpart III - Runtime Resources Layer</i>	173
15 Platform Package	177
15.1 Overview	177
15.2 Organization of the Platform Package	178
15.3 PlatformModel Class Diagram	179
15.3.1 PlatformModel Class	179
15.3.2 AbstractPlatformElement Class (abstract)	180
15.3.3 AbstractPlatformRelationship Class (abstract)	180
15.4 PlatformInheritances Class Diagram	181
15.5 PlatformResources Class Diagram	181
15.5.1 ResourceType Class	182
15.5.2 NamingResource Class	183
15.5.3 MarshalledResource Class	183
15.5.4 MessagingResource Class	183
15.5.5 FileResource Class	184
15.5.6 ExecutionResource Class	184
15.5.7 LockResource Class	184
15.5.8 StreamResource Class	184
15.5.9 DataManager Class	184
15.5.10 PlatformEvent Class	185
15.5.11 PlatformAction Class	185
15.5.12 ExternalActor Class	185
15.6 PlatformRelations Class Diagram	186
15.6.1 BindsTo Class	186
15.7 ProvisioningRelations Class Diagram	186
15.7.1 Requires Class	187

15.7.2 PlatformActions Class Diagram	187
15.7.3 ManagesResource Class	188
15.7.4 ReadsResource Class	189
15.7.5 WritesResource Class	189
15.7.6 DefinedBy Class	189
15.7.7 Deployment Class Diagram	190
15.7.8 DeployedComponent Class	191
15.7.9 DeployedSoftwareSystem Class	192
15.7.10 Machine Class	192
15.7.11 DeployedResource Class	193
15.7.12 RuntimeResources Class Diagram.....	193
15.7.13 RuntimeResource (generic)	194
15.7.14 Process Class	194
15.7.15 Thread Class	194
15.7.16 RuntimeActions Class Diagram	194
15.7.17 Loads Class	195
15.7.18 Spawns Class	196
15.7.19 ExtendedPlatformElements Class Diagram	196
15.7.20 PlatformElement Class (generic)	197
15.7.21 PlatformRelationship Class (generic)	197
16 UI Package	199
16.1 Overview	199
16.2 Organization of the UI Package	200
16.3 UIModel Class Diagram	200
16.3.1 UIModel Class	201
16.3.2 AbstractUIElement Class (abstract)	201
16.3.3 AbstractUIRelationship Class (abstract)	202
16.4 UIInheritances Class Diagram	202
16.5 UIResources Class Diagram	203
16.5.1 UIResource Class (generic)	204
16.5.2 UIDisplay Class (generic)	204
16.5.3 Screen Class	204
16.5.4 Report Class	205
16.5.5 UIField Class	205
16.5.6 UIEvent Class	205
16.5.7 UIAction Class	205
16.6 UIRelations Class Diagram	206
16.6.1 UIFlow Class	206
16.6.2 UILayout Class	207
16.7 UIActions Class Diagram	207
16.7.1 Displays Class	208
16.7.2 DisplaysImage Class	208
16.7.3 ManagesUI Class	208
16.7.4 ReadsUI Class	209

16.7.5 WritesUI Class	209
16.8 ExtendedUIElements Class Diagram	210
16.9 UIElement Class (generic)	210
16.9.1 UIRelationship Class (generic)	210
17 Event Package	213
17.1 Overview	213
17.2 Organization of the Event Package	214
17.3 EventModel Class Diagram	214
17.3.1 EventModel Class	215
17.3.2 AbstractEventElement Class (abstract)	215
17.3.3 AbstractEventRelationship Class (abstract)	216
17.4 EventResources Class Diagram	216
17.4.1 EventResource Class (generic)	217
17.4.2 Event Class	217
17.4.3 State Class	218
17.4.4 InitialState Class	218
17.4.5 Transition Class	218
17.4.6 OnEntry Class	218
17.4.7 OnExit Class	219
17.4.8 EventAction Class	219
17.5 EventRelations Class Diagram	219
17.5.1 NextState Class	220
17.6 ConsumesEvent Class	220
17.7 EventActions Class Diagram	220
17.7.1 ReadsState Class	221
17.7.2 ProducesEvent Class	222
17.7.3 HasState Class	222
17.8 ExtendedEventElements Class Diagram	222
17.8.1 EventElement Class (generic)	223
17.8.2 EventRelationship Class (generic)	223
18 Data Package	225
18.1 Overview	225
18.2 Organization of the Data Package	226
18.3 Data Model Class Diagram	226
18.3.1 DataModel Class	227
18.3.2 AbstractDataElement Class (abstract)	228
18.3.3 AbstractDataRelationship Class (abstract)	228
18.4 Data Inheritances Class Diagram	228

18.5	DataResources Class Diagram	229
18.5.1	DataResource Class (generic)	230
18.5.2	DataContainer Class (generic)	230
18.5.3	Catalog Class	231
18.5.4	RelationalSchema Class	231
18.5.5	DataEvent Class	232
18.5.6	DataAction Class	232
18.6	ColumnSet Class Diagram	233
18.6.1	ColumnSet (generic)	233
18.6.2	RelationalTable Class	234
18.6.3	RelationalView Class	237
18.6.4	DataSegment Class	237
18.6.5	RecordFile Class	239
18.7	KeyIndex Class Diagram	244
18.7.1	IndexElement Class (generic)	244
18.7.2	UniqueKey Class	245
18.7.3	ReferenceKey Class	245
18.7.4	Index Class	245
18.8	Key Relations Class Diagram	246
18.8.1	KeyRelationship Class	246
18.9	DataActions Class Diagram	247
18.9.1	ReadsColumnSet Class	247
18.9.2	WritesColumnSet Class	248
18.9.3	ManagesData Class	248
18.9.4	HasContent Class	249
18.10	StructuredData Class Diagram	254
18.10.1	XMLSchema	255
18.10.2	AbstractContentElement (abstract)	255
18.11	ContentElements Class Diagram	255
18.11.1	ContentItem (generic)	256
18.11.2	ComplexContentType	257
18.11.3	SimpleContentType	257
18.11.4	ContentRestriction	257
18.11.5	AllContent Class	260
18.11.6	SeqContent Class	260
18.11.7	ChoiceContent Class	260
18.11.8	GroupContent Class	261
18.11.9	MixedContent Class	261
18.11.10	ContentAttribute Class	261
18.11.11	ContentElement Class	261
18.11.12	ContentReference Class	261
18.12	ContentRelations Class Diagram	266
18.12.1	TypedBy Class	267
18.12.2	DatatypeOf Class	268

18.12.3 ReferenceTo Class	268
18.12.4 ExtensionTo Class	268
18.12.5 RestrictionOf Class	269
18.13 ExtendedDataElements Class Diagram	269
18.13.1 ExtendedDataElement Class	270
18.13.2 DataRelationship Class	270
<i>Subpart IV - Abstractions Layer</i>	273
19 Structure Package	275
19.1 Overview	275
19.2 Organization of the Structure Package	276
19.3 StructureModel Class Diagram	276
19.3.1 StructureModel Class	277
19.3.2 AbstractStructureElement Class (abstract)	277
19.3.3 AbstractStructureRelationship Class (abstract)	278
19.3.4 Subsystem Class	278
19.3.5 Layer Class	278
19.3.6 Component Class	278
19.3.7 SoftwareSystem Class	279
19.3.8 ArchitectureView Class	279
19.4 StructureInheritances Class Diagram	279
19.5 ExtendedStructureElements Class Diagram	280
19.5.1 StructureElement Class (generic)	280
19.5.2 StructureRelationship Class (generic)	280
20 Conceptual Package	283
20.1 Overview	283
20.2 Organization of the Conceptual Package	285
20.3 ConceptualModel Class Diagram	285
20.3.1 ConceptualModel	286
20.3.2 AbstractConceptualElement (abstract)	287
20.3.3 AbstractConceptualRelationship Class (abstract)	288
20.4 ConceptualInheritances Class Diagram	288
20.5 ConceptualElements Class Diagram	288
20.5.1 ConceptualContainer Class	289
20.5.2 TermUnit	290
20.5.3 FactUnit	290
20.5.4 RuleUnit	290
20.5.5 ConceptualRole	291
20.5.6 BehaviorUnit Class	291
20.5.7 ScenarioUnit Class	291

20.6 ConceptualRelations Class Diagram	292
20.6.1 ConceptualFlow Class	292
20.7 ExtendedConceptualElements Class Diagram	299
20.7.1 ConceptualElement Class (generic)	300
20.7.2 ConceptualRelationship Class (generic)	300
21 Build Package	303
21.1 Overview	303
21.2 Organization of the Build Package	303
21.3 BuildModel Class Diagram	304
21.4 BuildModel Class	304
21.4.1 AbstractBuildElement Class (abstract)	305
21.4.2 AbstractBuildRelationship Class (abstract)	305
21.4.3 Supplier Class	305
21.4.4 Tool Class	305
21.4.5 SymbolicLink Class	306
21.5 BuildInheritances Class Diagram	306
21.6 BuildResources Class Diagram	306
21.6.1 BuildResource Class	307
21.6.2 BuildComponent Class	308
21.6.3 BuildDescription Class	308
21.6.4 BuildStep Class	308
21.7 BuildRelations Class Diagram	308
21.7.1 LinksTo Class	309
21.7.2 Consumes Class	310
21.7.3 Produces Class	310
21.7.4 SupportedBy Class	311
21.7.5 SuppliedBy Class	311
21.7.6 DescribedBy Class	312
21.8 ExtendedBuildElements Class Diagram	313
21.8.1 BuildElement Class (generic)	313
21.8.2 BuildRelationship Class (generic)	314
Annex A - Semantics of the Micro KDM Action Elements	315
Index	327

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19500-1 was prepared by Technical Committee ISO/IEC JTC1, Information technology, in collaboration with the Object Management Group (OMG), following the submission and processing as a Publicly Available Specification (PAS) of the OMG Common Object Request Broker Architecture (CORBA) specification Part 1 Version 3.1 CORBA Interfaces.

ISO/IEC 19506 is related to:

- ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1995, Information Technology - Open Distributed Processing - Reference Model: Foundations
- ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1995, Information Technology - Open Distributed Processing - Reference Model: Architecture
- ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1997, Information Technology - Open Distributed Processing - Interface Definition Language

ISO/IEC 19506 consists of the following, under the general title *Information technology - Open distributed processing - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM)*.

Apart from this Foreword, the text of this International Standard is identical with that for the OMG specification for Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.3.

Introduction

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability and portability can be integrated.

RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. The scopes and objectives of the RM-ODP Part 2 and the UML, while related, are not the same and, in a number of cases, the RM-ODP Part 2 and the UML specification use the same term for concepts which are related but not identical (e.g., interface). Nevertheless, a specification using the Part 2 modeling concepts can be expressed using UML with appropriate extensions (using stereotypes, tags, and constraints).

RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2. Given the relation between UML as a modeling language and Part 3 of the RM-ODP standard, it is easy to show that UML is suitable as a notation for the individual viewpoint specifications defined by the RM-ODP.

This International Standard for Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) is a standard for the technology specification of an ODP system. It defines a technology to provide the infrastructure required to support functional distribution of an ODP system, specifying functions required to manage physical distribution, communications, processing and storage, and the roles of different technology objects in supporting those functions.

Information technology - Object Management Group Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM)

1 Scope

This International Standard defines a meta-model for representing *existing software assets*, their 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 assets* (for example, source code modules, database descriptions, build scripts, etc.) to obtain explicit knowledge. Each tool produces a portion of the knowledge about existing software assets. Such tool-specific knowledge may be implicit (“hard-coded” in the tool), 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 repository structure that facilitates the exchange of data contained within individual tool models that represent *existing software assets*. The meta-model represents the physical and logical assets at various levels of abstraction. The primary purpose of this meta-model is to provide a common interchange format that will allow interoperability between existing modernization and software assurance tools, services, and their respective intermediate representations.

2 Conformance and Compliance

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 by allowing integration information about a complex *enterprise application* from multiple sources, 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 International Standard 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. Consequently, the definition of compliance for KDM requires a balance to be drawn between modularity and ease of interchange. 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 consists of a single KDM package that defines meta-model elements to represent particular aspects of the system under study. KDM domains correspond to the well-known concept of *architecture views*. For example, the Structure domain enables users to discover architectural elements of source code from the system under study, while the Business Rules domain provides users with behavioral elements of the same system such as features or process rules.

The following domains of knowledge have been identified as the foundation for defining compliance in KDM: 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.

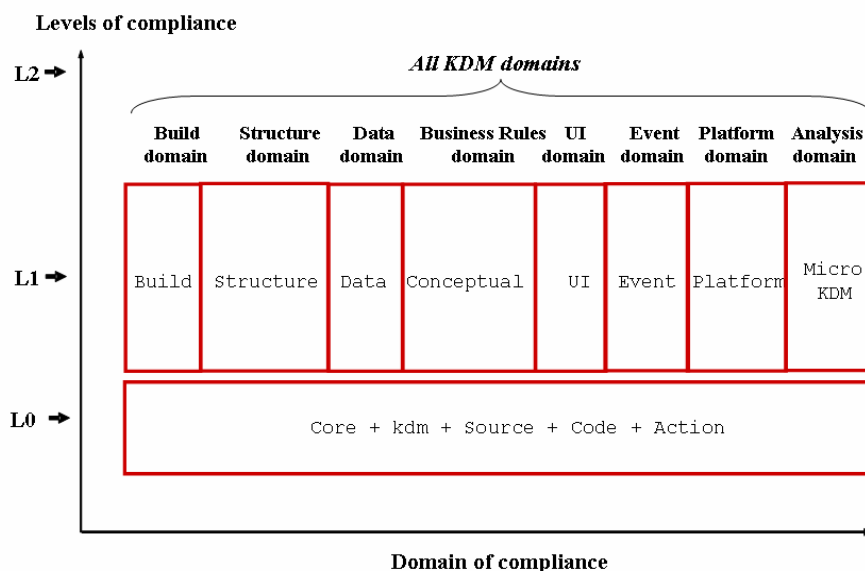


Figure 2.1- Domains and levels of KDM compliance

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 two KDM compliance levels:

- Level 0 (L0) - This compliance level contains 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 must completely support all model elements within all packages for L0 level.

- Level 1 (L1) - This level addresses KDM domains and extends the capabilities provided by Level 0. Specifically, it adds 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 must completely support all model elements defined by the package for that domain and satisfy all semantic constraints specified for that domain.

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

2.3 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		Import KDM models based on complete KDM XMI schema into existing tool; support specified mapping between KDM and existing model in 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 traceability to the physical artifacts of the application from Source package.	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.	Provide capability to analyze artifacts of an application 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	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Structure package.	L0 compliance for import; Support KDM API as defined by the Structure package.	L0 compliance for export; Provide capability to analyze architecture components of existing application and generate KDM Structure model according to Structure package.
	DATA	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Data package.	L0 compliance for import; Support KDM API as defined by the Data package.	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	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Platform package.	L0 compliance for import; Support KDM API as defined by the Platform and Runtime packages.	L0 compliance for export; Provide capability to analyze platform artifacts for specified platform and generate KDM Platform model according to Platform package.
	BUILD	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Build package.	L0 compliance for import; Support KDM API as defined by the Build package.	L0 compliance for export; Provide capability to analyze build artifacts for specified build environment and generate KDM Build model according to Build package.

Table 2.1- Compliance Statements

	UI	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the UI package.	L0 compliance for import; Support KDM API as defined by the UI package.	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	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Event package.	L0 compliance for import; Support KDM API as defined by the Event package.	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	L0 compliance for analysis; extend operations of existing tool to support meta-model elements of the Conceptual package.	L0 compliance for import; Support KDM API as defined by the Conceptual package.	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	L0 compliance for analysis; extend operations of existing tool to support micro KDM actions as specified in sub clause 14 micro KDM and Annex A.	L0 compliance for import; Support micro KDM actions as specified in sub clause 14 micro KDM and Annex A.	L0 compliance for export; Provide capability to analyze artifacts of existing application to the level of detail specified in sub clause 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
L2		L0 import compliance for analysis; L1 import-analysis compliance for all KDM domains.	L0 compliance for import; Support KDM API as defined by all KDM packages.	L0 export compliance; 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 International Standard. For dated references, subsequent amendments to or revisions of any of these publications do not apply.

- OMG UML Infrastructure Specification, v2.3, formal/2010-05-03
- OMG Meta-Object Facility (MOF) Specification, v2.0, formal/2006-01-01
- OMG MOF XML Metadata Interchange (XMI) Specification, v2.1, formal/2005-09-01
- OMG Semantics of Business Vocabularies and Business Rules (SBVR) Specification, v1.0, formal/2008-01-02
- ISO/IEC 11404:2007 Information technology -- General Purpose Datatypes (GPD)

4 Terms and Definitions

This subclause contains only those terms which are used in a specialized way throughout the KDM specification. The majority of terms in KDM are used either according to their accepted dictionary definitions or according to commonly accepted definitions that may be found in ISO glossaries or other well-known collections of software engineering terms. Some combinations of common terms used in KDM, while not meriting glossary definition, are explained for clarity in the context where they are used.

Abstraction: A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information.

Aggregation: a derived relationship between two elements that are groups of other elements that represents all individual relationships between the grouped elements of the two groups.

Architecture-Driven Modernization (ADM): ADM is the process of understanding and evolving existing software assets of a system of interest. ADM focuses at collecting, sharing, utilizing, transforming, presenting, maintaining and storing models of the architectural aspects of existing systems. ADM does not preclude source-to-source migrations (where appropriate), but encourages user organizations to consider modernization from an analysis and design perspective. In doing so, project teams ensure that obsolete concepts or designs are not propagated into modern languages and platforms.

Build: An operational version of a system or component that incorporates a specified subset of the capabilities that the final product provides.

Build process: a process of transforming of project code base into usable applications. The end result of a software build is a collection of files that constitute a product in a distributable package. In this case, package can mean a standalone application, Web service, compact disc, hotfix, or bug fix. Each step of a build process is a transformation performed by software running on a general purpose computer. A simple software build may involve compiling a single source code file into an executable code. A complex software build may involve orchestrating hundreds of versions of thousands of files with millions of lines of source code such that a correct executable code results from the compilation. The implementation of a system also involves deploying the build onto the system nodes, and applying appropriate configuration settings.

Component: a functionally or logically distinct part of a system. A component may be hardware or software and may be subdivided into other components. Often a component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. Such component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

Container: a model element that owns one or more distinct elements through the special “owns” (“contains”) relationships between the container element and owned elements. “Containment” relationships form a special group of the corresponding owned elements. No element has more than one container.

Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

Element: one of the parts of a compound or complex whole. For example, a model element, a meta-model element.

Group: a number of model elements regarded as a unit formed by traceability relationships to a single distinct element. An element may be part of multiple groups, including a single group formed by the “containment” relationships between a container and its owned elements. An element is said to group together one or more elements, if these elements have traceability relationships to the element.

Hierarchy: an arrangement of model elements according to traceability relationships, where an element that “owns” or “group” other elements is considered at a higher level than the owned (grouped) elements.

Interface: (1) a shared boundary or connection between two dissimilar objects, devices or systems through which information is passed. The connection can be either physical or logical. (2) a named set of operations that characterize the behavior of an entity

Item: that which can be individually described or considered. See also Component, Element, Unit, Module.

KDM Entity: a meta-model element (as well as the corresponding model elements) that represents a thing of significance of the system of interest, about which information needs to be known or held. A KDM entity is an abstraction of some element of the system of interest that has a distinct, separate existence objective or conceptual reality, a self-contained piece of data that can be referenced as a unit. As a model element each KDM entity is an instance of some specific metamodel element and it is usually the endpoint of distinct KDM relationships.

KDM instance: a collection of KDM model elements that represent one or more views of the system of interest.

KDM model: a meta-model element (as well as the corresponding model elements) that is a container for a KDM view.

KDM Relationship: a meta-model element (as well as the corresponding model elements) that represents some semantic association between elements of the system of interest. All KDM relationships are binary. As a model element each KDM relationship is an instance of some specific meta-model element.

Meta-model: a special kind of model that specifies the abstract syntax of a modeling language. The typical role of a metamodel is to define the semantics for how model elements in a model get instantiated. A model typically contains model elements. These are created by instantiating model elements from a metamodel (i.e., metamodel elements).

Meta-model element: an element of a meta-model from which model elements are instantiated.

Model: a model represents a system of interest, from the perspective of a related set of concerns. The model is related to the system by an explicit or implicit isomorphism. Models are instances of MOF meta-models and therefore consist of model elements and links between them.

Model element: instance of a meta-model element.

Module: (1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. (2) A logically separable part of a program.

Resource: any physical or virtual component of limited availability within a computer system available for a given purpose and managed by the runtime platform.

Runtime platform: the set of hardware and software components that implement the services utilized by the application software. A runtime system generally controls how several tasks are scheduled to run, and manages resources. Its provisions for the programmer typically form an Application Programming Interface- a set the well-documented ways of using the system.

Segment: A collection of data that corresponds to one or more coherent views of a system of interest that is stored or transferred as a unit.

Software artifact: A software artifact is a tangible machine-readable document created during software development. Examples are requirement specification documents, design documents, source code and executables.

Software asset: A software asset is a description of a partial solution (such as a component or design document) or knowledge (such as requirements database or test procedures) that engineers use to build or modify software products. A software asset is a set of one or more related artifacts that have been created or harvested for the purpose of applying that asset repeatedly in subsequent contexts. The asset consumer is an architect or a designer of any type of IT or business process solutions from solution business modeling, analysis (assets used are models) and design to application development (assets used are pieces of code).

Traceability: The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match.

Unit: (1) a piece or complex of apparatus serving to perform one particular function (2) A software element that is not subdivided into other elements.

User interface: An interface that enables information to be passed between a human user and hardware or software components of a computer system.

View: A representation of a whole system from the perspective of a related set of concerns.

Viewpoint: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

5 Symbols

There are no symbols defined.

6 Additional Information

6.1 Changes to Other OMG Specifications

There are no changes to other OMG specifications.

6.2 How to Read this International Standard

The rest of this International Standard contains the technical content of KDM.

Clause 7. Specification overview - Provides design rationale for KDM.

Clause 8. KDM - Gives the overview of the packages of KDM.

Part I - The KDM Infrastructure Layer

Clause 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.

Clause 10. KDM package - Describes 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 models. This package determines the queries against KDM models.

Clause 11. Source package - This package describes meta-model elements for specifying the linkage between the KDM model artifacts and their physical implementations in the artifacts of existing software. Elements of the Source package allow viewing the source code, corresponding to KDM model elements.

Part II - The Program Elements Layer

Clause 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.

Clause 13. Action package - Describes 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.

Clause 14. Micro KDM - Describes the guidelines and constraints for semantically precise KDM representations.

Part III - The Runtime Resources Layer

Clause 15. Platform package - Describes the meta-model elements for representing 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.

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

Clause 17. Event package - Describes meta-model elements that represent basic elements related to behavior of applications in terms of events, messages and responses.

Clause 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

Clause 19. Structure package - Describes the meta-model elements for representing the logical organization of the software system in terms of logical subsystems, architectural layers, components and packages.

Clause 20. Conceptual package - Describes the meta-model elements for representing business domain knowledge about existing applications in the context of other KDM views.

Clause 21. Build package - Describes the meta-model elements for representing the artifacts involved in building the software system (the engineering view of the software system).

7 Overview

This International Standard 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, which formalize the *viewpoint language* for the domain. KDM focuses at 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.

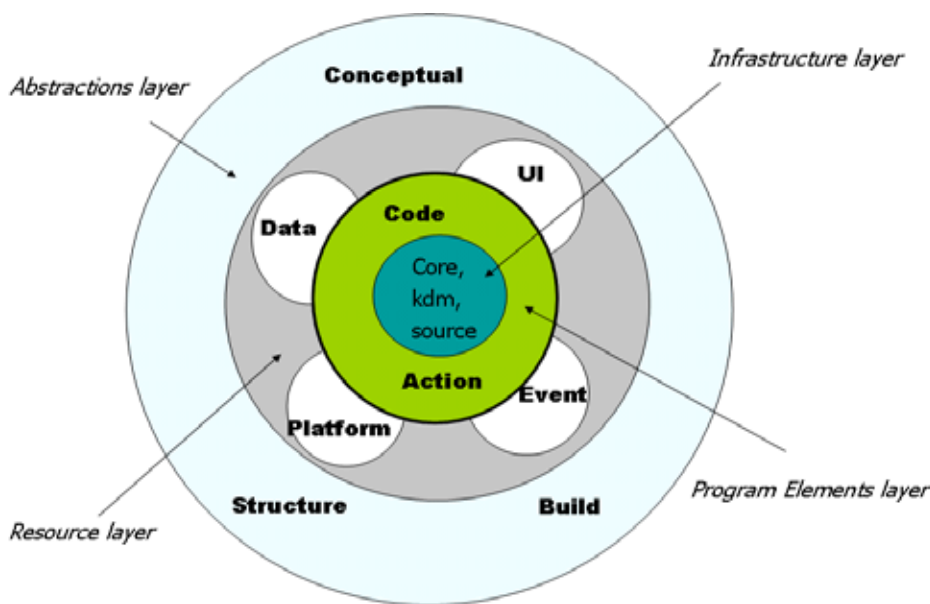


Figure 7.1- Layers, packages, and separation of concerns in KDM

KDM is organized into the following 4 layers:

- 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 Infrastructure Layer consists of the following 3 packages: Core, “kdm”, and Source. Core package and the package named “kdm” 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 Resources 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 Entry-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.
- KDM provides model refactoring capabilities. For example, a KDM tool can support moving entities between containers and map changes in the model to changes in the code 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).

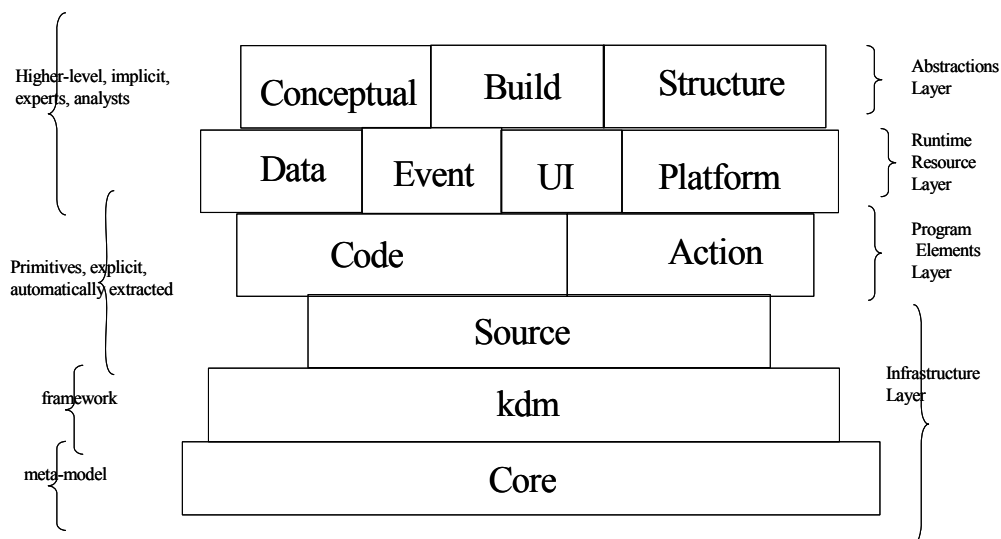


Figure 8.1 - Structure of KDM Packages

The KDM specification contains 12 packages; each package is defined by one or more class diagrams.

The Core package defines the basic meta-elements (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 package depends on the Core package. The nature of this dependency is that the meta-model elements defined by each package are subclasses of one of the meta-model elements defined in the Core package. Also, each package depends on the package with name “kdm.” Each KDM package above the package with name “kdm” in Figure 8.1 defines a KDM model, which corresponds to a certain facet of knowledge about an existing software system. The package with name “kdm” provides the infrastructure for all KDM models.

The nature of the dependency on the package with name “kdm” is as follows:

- First, each package defines a subclass of the KDMModel class, defined in that package.
- Second, each package provides several concrete classes that are instantiated in each KDM instance as part of the infrastructure.

- Third, the package with name “kdm” 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 corresponding meta-model elements 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 package with name “kdm” provides shared context for 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”).

Subpart I - 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 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 classes defined in the Core package. 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.

The Core package 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 package with name “kdm” defines several elements that together constitute the *framework* of each KDM instance. 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 views are 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 specification 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 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.

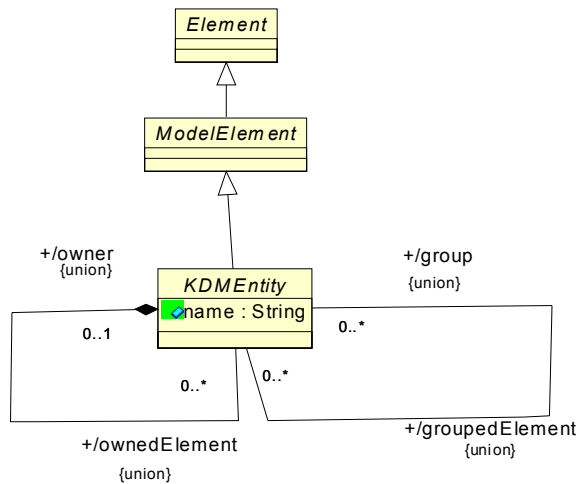


Figure 9.1- CoreEntities Class Diagram

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 ModelElement 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 Infrastructure Layer.

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

9.3.3 KDMEntity Class (abstract)

A KDMEntity 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 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 element 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.

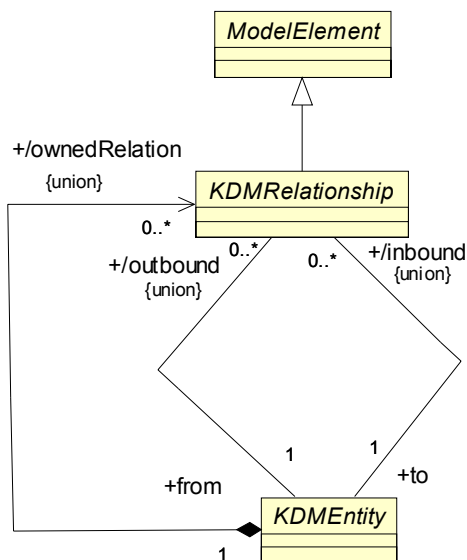


Figure 9.2 - CoreRelations Class Diagram

9.4.1 KDMRelationship Class (abstract)

A **KDMRelationship** 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 <i>target</i> 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 “subsets” mechanism. Concrete properties redefine the properties of the parent classes, defined in the Core package.
from:KDMEntity[1]	The <i>origin</i> 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 “subsets” 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

KDMRelationship meta-model element is an abstract element. The concrete KDM relationships between KDM entities in KDM views are instances of concrete subclasses of KDMRelationship. Each instance of KDMRelationship has exactly one target and exactly one origin. Each concrete subclass of KDMRelationship defines the acceptable types of its endpoints.

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.

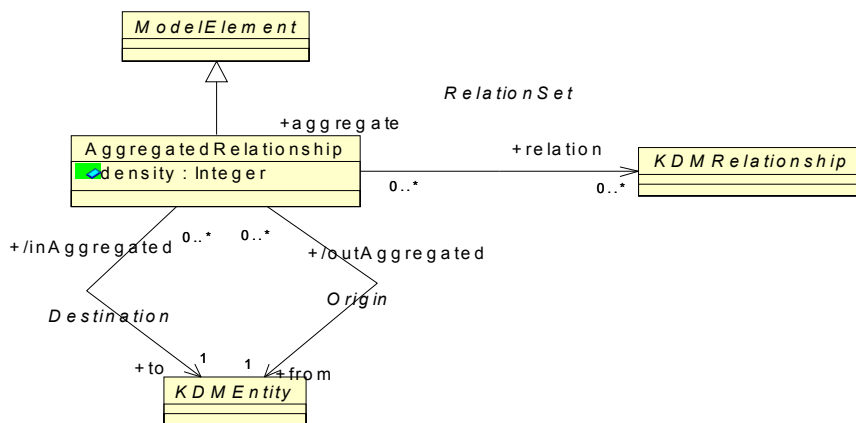


Figure 9.3 - AggregatedRelations Class Diagram

9.5.1 AggregatedRelationship Class

The set of AggregatedRelationship elements 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 relationships 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 relationships represented by the given aggregated relationship.

Semantics

AggregatedRelationships is 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. AggregatedRelationship 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 relationship “ x in* C ” means that x is in container C or in some sub-container of C , transitively.

For relationship R , let R' be the corresponding aggregated relationship.

Given containers $C1$ and $C2$ and the relationship R , let

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

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

Then

$$C1 R' C2 \text{ iff } |P| > 0$$

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

The density of $C1 \text{ ‘} C2$ is then simply $|P|$, the size of the set P .

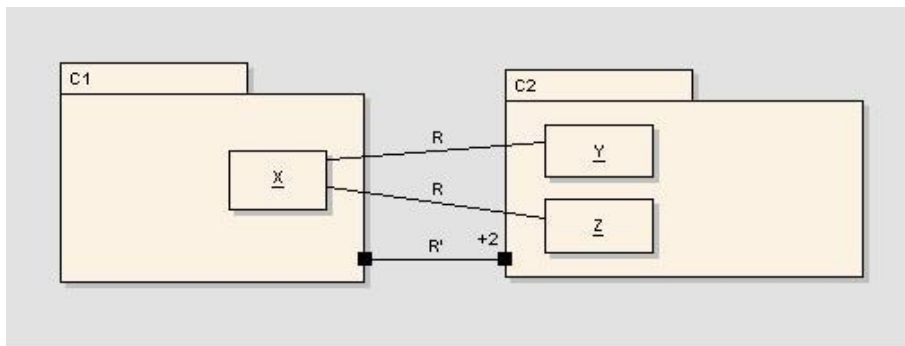


Figure 9.4 - AggregatedRelationships illustrated

Figure 9.4 illustrates Containers and aggregated relationships. It uses the following notation. UML package symbols $C1$ and $C2$ represent KDM Containers, UML associations represent KDMRelations. An arrow at the end of an association indicates the direction of the relationship, when there are no arrows at either end of the association (as in the Figure 9.4), this indicates two relationships, one in each direction. The numbers at the ends of associations, such as “+2”, represent the density of the corresponding KDM relationship. The KDM density has a different interpretation than UML multiplicity: since KDM represents an existing application, the exact relationships 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 relationships are collections of more primitive relationships, which at the end are some basic code facts, for example “procedure x calls procedure y .” Such basic fact has density 1. A primitive code relationship 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 relationship between modules A and B with density 2 ($2=1+1$). In this case, the aggregated relationship represents the collection of the two primitive relationships between modules A and B.

9.5.2 KDMEntity (additional properties)

Operations

<code>createAggregation(otherEntity:KDMEntity)</code>	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).
<code>deleteAggregation (aggregatedRelation:AggregatedRelationship)</code>	This operation deletes the given aggregated relationship.
<code>getInAggregated():AggregatedRelationship[0..*]</code>	This operation returns the set of AggregatedRelationship for which the target is the current KDM Entity.
<code>getOutAggregated():AggregatedRelationship[0..*]</code>	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.

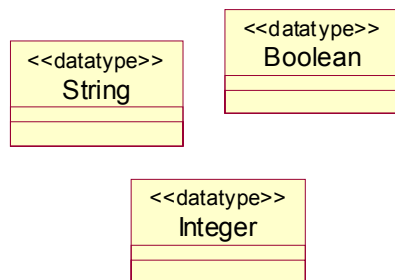


Figure 9.5 - Datatypes Class Diagram

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 The Package Named “kdm”

10.1 Overview

The package named “kdm” defines the key infrastructure elements that determine patterns for constructing KDM views of existing software systems. KDM views (also referred to as KDM instances) are collections of the elements that are instances of the meta-model elements defined by the KDM specification, 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 International Standard. The package named “kdm” 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 Infrastructure Layer together with the elements defined in the Source package.

10.2 Organization of the KDM Framework

The package with name “kdm” 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 organized 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. On the other hand, the implementers of KDM import tools should not make any assumptions about the organization of KDM model elements into models or organization of models into 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 package with name “kdm” consists of the following 5 class diagrams:

- Framework – defines the basic elements of the KDM framework.
- Audit – defines audit information for KDM model elements.
- Annotations - provides user-defined attributes and annotations to the modeling elements.
- Extensions - a class diagram that defines the overall organization of the light-weight extension mechanism of KDM.
- ExtendedValues - the tagged values used by the light-weight extension mechanism.

The package with name “kdm” 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.

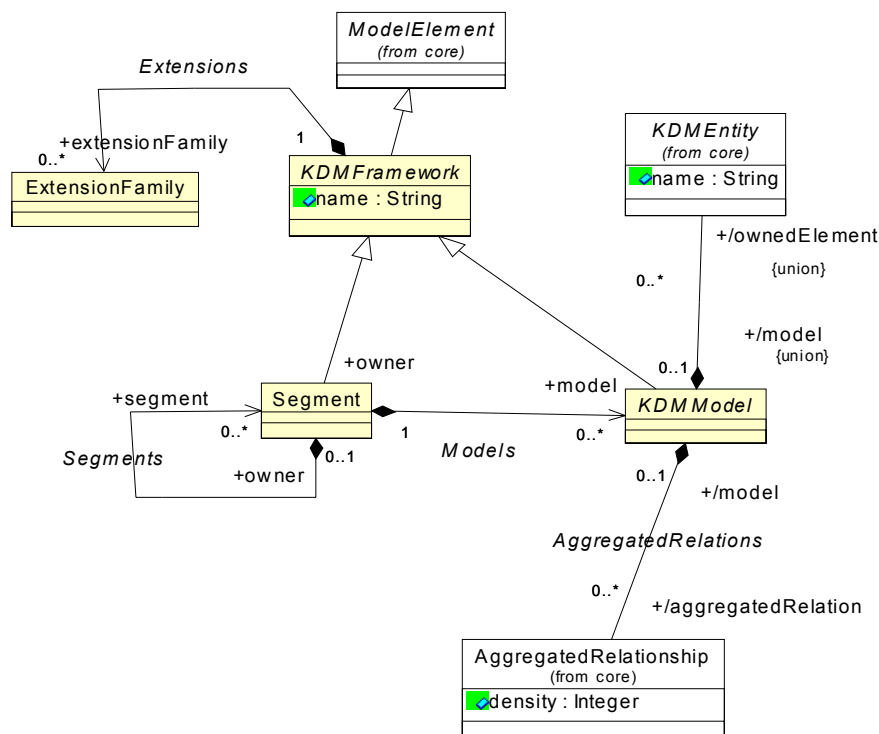


Figure 10.1 - Framework Class Diagram

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. These elements are contains for KDM light-weight extensions (extension property). The KDM extension mechanism is described further in this clause.

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

10.3.2 KDMModel Class (abstract)

A KDMModel is an abstract class that defines common properties of KDM model instances which 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 define 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.

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.