Architecture-driven Modernization (ADM): Knowledge Discovery Meta-model (KDM) Specification

This OMG document replaces the submission document (ad/06-03-01) and the Draft Adopted specification (ptc/06-05-02). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by September 4, 2006.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*.

The FTF Recommendation and Report for this specification will be published on December 18, 2006. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG Adopted Specification
ptc/06-06-07

Date: June 2006

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

OMG Adopted Specification
ptc/06-06-07

Copyright © 2006, Allen Systems Group, Inc.
Copyright © 2006, EDS
Copyright © 2006, Flashline
Copyright © 2006, IBM
Copyright © 2006, KDM Analytics
Copyright © 2006, Klocwork, Inc.
Copyright © 1997-2006, Object Management Group.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

LICENSES

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

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

PATENTS

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

GENERAL USE RESTRICTIONS

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

DISCLAIMER OF WARRANTY

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

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

RESTRICTED RIGHTS LEGEND

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

TRADEMARKS

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

COMPLIANCE

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

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

OMG's Issue Reporting Procedure

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

Table of Contents

Preface xiii

1 Scope 1

2 Conformance 1

2.1 KDM Domains 1

2.2 Compliance Levels 2

2.2.1 Meaning and Types of Compliance 3

3 Normative References 5

4 Terms and Definitions 5

5 Symbols 5

6 Additional Information 5

6.1 Changes to Other OMG Specifications 5

6.2 How to Read this Specification 5

6.3 Acknowledgements 7

7 Specification Overview 9

8 KDM 11

8.1 Overview 11

8.2 Organization of the KDM Packages 11

9 Core Package 13

9.1 Overview 3

9.2 Organization of the Core Package 13

9.3 CoreEntities Class Diagram 13

9.3.1 Element Class (Abstract) 14
9.3.2 ModelElement Class (Abstract) 14
9.3.3 KDMEntity Class (Abstract) 14
9.3.4 KDMContainer Class (Abstract) 15
9.3.5 KDMGroup Class (Abstract) 16

9.4 CoreRelations Class Diagram 16

9.4.1 KDMRelationship Class (Abstract) 17

9.4.2 KDMEntity (additional properties) 18

9.5 AggregateRelations Class Diagram 18

9.5.1 KDMAggregatedRelationship Class 19
9.5.2 KDMEntity (additional properties) 21

9.6 Datatypes Class Diagram 21

9.6.1 InstanceKind Data Type (enumeration) 21
9.6.2 Boolean Type (datatype) 22
9.6.3 String Type (datatype) 22
9.6.4 Integer Type (datatype) 22

9.7 Extensions Class Diagram 22

9.7.1 Stereotype Class 23
9.7.2 TaggedDefinition Class 24
9.7.3 ExtensionFamily Class 24
9.7.4 TaggedValue Class 25
9.7.5 ModelElement (additional properties) 26

9.8 Annotations Class Diagram 26

9.8.1 Attribute Class 27
9.8.2 Annotation Class 27
9.8.3 Element (additional properties) 28

10 KDM Package 29

10.1 Overview 29

10.2 Organization of the Kdm Package 29

10.3 Framework Class Diagram 30

10.3.1 KDMFramework Class (abstract) 30
10.3.2 KDMModel Class (Abstract) 31
10.3.3 KDMSegment Class 31
10.3.4 KDMRoot Class 32

10.4 ModelRoot Class Diagram 32

10.4.1 ModelRoot Class (abstract) 33

10.5 Audit Class Diagram 33

10.5.1 Audit Class 34
10.5.2 KDMFramework (additional properties) 34

11 Source Package 35

11.1 Overview 35

11.2 Organization of the Source Package 35

11.3 SourceRef Class Diagram 35

11.3.1 SourceRef Class 36
11.3.2 CodeElement (additional properties) 37
11.3.3 DataElement (additional properties) 37

11.3.4 UIElement (additional properties) 37
11.4 SourceRegion Class Diagram 37
11.4.1 SourceRegion Class 38
12 Code Package 41
12.1 Overview 41
12.2 Organization of the Code Package 41
12.3 CodeInheritances Class Diagram 41
12.4 CodeModel Class Diagram 42
12.4.1 CodeModel Class 43
12.4.2 CodeElement Class (abstract) 44
12.4.3 CodeResource Class (abstract) 44
12.4.4 CodeGroup Class 44
12.4.5 CodeContainer Class (abstract) 45
12.4.6 TypeElement Class (abstract) 45
12.4.7 TypeContainer Class (abstract) 45
12.5 CodeRelations Class Diagram 46
12.5.1 InterfaceRelationship Class (abstract) 46
12.5.2 TemplateRelationship Class 47
12.5.3 TypeRelationship Class 47
12.5.4 PrototypeRelationship Class 47
12.6 CallableUnits Class Diagram 47
12.6.1 CallableElement Class 48
12.6.2 CallableUnit Class 48
12.6.3 BlockUnit Class 49
12.6.4 MethodUnit Class 49
12.6.5 ConstructorUnit Class 49
12.6.6 OperatorUnit Class 49
12.7 Module Class Diagram 49
12.7.1 Module Class 50
12.7.2 CompilationUnit Class 50
12.7.3 SharedUnit Class 51
12.7.4 CodeAssembly Class 51
12.8 Prototype Class Diagram 51
12.8.1 PrototypeUnit Class 51
12.8.2 PrototypedBy Class 52
12.8.3 CodeElement Class (additional properties) 52
12.9 Macro Class Diagram 52
12.9.1 MacroUnit Class 53
12.10 Template Class Diagram 53
12.10.1 TemplateUnit Class 54

12.10.2 TemplateParameter Class 54
12.10.3 TemplateInstance Class 55

12.11 TemplateRelations Class Diagram 55
12.11.1 Instantiates Class 56
12.11.2 InstanceOf Class 56
12.11.3 CodeResource (additional properties) 57

12.12 SimpleTypes Class Diagram 57
12.12.1 SimpleTypeUnit Class 58
12.12.2 NamedTypeUnit Class 58
12.12.3 PredefinedTypeElement Class 58

12.13 PredefinedTypes Class Diagram 58
12.13.1 StringUnit Class 59
12.13.2 IntegerUnit Class 59
12.13.3 CharUnit Class 59
12.13.4 BooleanUnit Class 59
12.13.5 FloatUnit Class 60
12.13.6 FixedPointUnit Class 60
12.13.7 DecimalUnit Class 60
12.13.8 DateUnit Class 60
12.13.9 TimeUnit Class 60

12.14 DerivedTypes Class Diagram 61
12.14.1 DerivedTypeElement Class 61
12.14.2 RefinementType Class 62
12.14.3 PointerType Class 62
12.14.4 ArrayType Class 62

12.15 EnumerationTypes Class Diagram 62
12.15.1 EnumeratedUnit Class 63
12.15.2 EnumeratedLiteral Class 63

12.16 CompositeTypes Class Diagram 64
12.16.1 CompositeTypeElement Class 64
12.16.2 UnionUnit Class 64
12.16.3 CompositeUnit Class 65

12.17 ClassTypes Class Diagram 65
12.17.1 ClassUnit Class 65

12.18 Signature Class Diagram 66
12.18.1 Signature Class 66

12.19 Interface Class Diagram 67
12.19.1 Interface Class 67

12.20 InterfaceRelations Class Diagram 68
12.20.1 Implements Class 68
12.20.2 ImplementationOf Class 69
12.20.3 CodeResource (additional properties) 69

12.20.4 Interface (additional properties) ... 69
12.20.5 Signature (additional properties) ... 70

12.21 TypeRelations Class Diagram ... 70
12.21.1 HasType Class ... 70
12.21.2 CodeResource (additional properties) ... 71
12.21.3 TypeElement (additional properties) ... 71

12.22 ClassRelations Class Diagram ... 71
12.22.1 Extends Class ... 72
12.22.2 TypeElement (additional properties) ... 72

12.23 Comment Class Diagram ... 72
12.23.1 CommentUnit Class ... 73

12.24 Visibility Class Diagram ... 73
12.24.1 VisibleIn Class ... 74
12.24.2 Namespace Class ... 74
12.24.3 CodeResource (additional properties) ... 74

13 Action Package ... 75

13.1 Overview ... 75

13.2 Organization of the Action Package ... 75

13.3 ActionRelations Class Diagram ... 75
13.3.1 ActionRelationship Class (abstract)... 76
13.3.2 FlowRelationship Class (abstract) ... 76
13.3.3 MacroRelationship Class (abstract) ... 76
13.3.4 CallableRelationship Class (abstract) ... 77
13.3.5 DataRelationship Class (abstract) ... 77
13.3.6 ImportRelationship Class (abstract) ... 77

13.4 ActionModel Class Diagram ... 77
13.4.1 ActionElement Class ... 78
13.4.2 ActionGroup Class ... 78
13.4.3 CallableElement (additional properties)... 79
13.4.4 CodeModel (additional properties) ... 79

13.5 ActionFlow Class Diagram... 79
13.5.1 ControlFlow Class ... 80
13.5.2 EntryFlow Class ... 80
13.5.3 ActionElement Class (additional properties) ... 81
13.5.4 CallableElement Class (additional properties) ... 81
13.5.5 Flow Class (abstract) ... 81
13.5.6 TrueFlow Class (abstract) ... 82
13.5.7 FalseFlow Class (abstract) ... 82
13.5.8 GuardedFlow Class (abstract) ... 82

13.6 CallableRelations Class Diagram ... 82
13.6.1 Calls Class ... 83

13.6.2 UsesCallable Class 84
13.6.3 Invokes Class 84
13.6.4 CallableElement (additional properties) 84
13.6.5 ActionElement Class (additional properties) 85
13.6.6 NamedTypeElement (additional properties) 85

13.7 DataRelations Class Diagram 85

13.7.1 Reads Class 86
13.7.2 Writes Class 86
13.7.3 UsesData Class 87
13.7.4 Creates Class 87
13.7.5 Destroys Class 88
13.7.6 Initializes Class 88
13.7.7 StorableElement (additional properties) 88
13.7.8 ActionElement Class (additional properties) 89

13.8 PrototypeRelations Class Diagram 89

13.8.1 UsesPrototype Class 90
13.8.2 ActionElement Class (additional properties) 90
13.8.3 PrototypeUnit Class (additional properties) 91

13.9 ImportRelations Class Diagram 91

13.9.1 ImportDirective Class 91
13.9.2 Imports Class 92
13.9.3 CodeResource Class (additional properties) 92

13.10 TypeRelations Class Diagram 92

13.10.1 UsesType Class 93
13.10.2 ActionElement Class (additional properties) 93
13.10.3 TypeElement Class (additional properties) 94

13.11 MacroRelations Class Diagram 94

13.11.1 Expands Class 94
13.11.2 MacroUnit Class (additional properties) 95

14 Build Package 97

14.1 Overview 97

14.2 Organization of the Build Package 97

14.3 BuildInheritances Class Diagram 97

14.4 BuildModel Class Diagram 98

14.4.1 BuildModel Class 99
14.4.2 BuildElement Class 99
14.4.3 BuildGroup Class 100
14.4.4 BuildResource Class 100
14.4.5 Directory Class 101
14.4.6 Origin Class 101
14.4.7 Tool Class 101

14.5 BuildResources Class Diagram 101
14.5.1 BuildResource Class (additional properties) 102
14.5.2 SourceFile Class 102
14.5.3 IntermediateFile Class 102
14.5.4 BuildComponent Class 103
14.5.5 BuildDescription Class 103
14.5.6 SymbolicLink Class 103
14.5.7 Image Class 103
14.6 BuildRelations Class Diagram 103
14.6.1 BuildRelationship Class (abstract) 104
14.6.2 LinksTo Class 104
14.6.3 DependsOn Class 105
14.6.4 GeneratedBy Class 105
14.6.5 BuildElement (additional properties) 105
14.6.6 SymbolicLink (additional properties) 106
15 Data Package 107
15.1 Overview 107
15.2 Organization of the Data Package 107
15.3 Data Inheritance 107
15.4 Data Model Class Diagram 108
15.4.1 DataModel Class 109
15.4.2 DataElement Class 109
15.4.3 DataGroup Class 110
15.4.4 DataContainer Class 110
15.5 KeyIndex Class Diagram 111
15.5.1 Index Class 111
15.5.2 UniqueKey Class 111
15.5.3 ReferenceKey Class 112
15.6 RelationalData Class Diagram 112
15.6.1 Catalog Class 113
15.6.2 DBSchema Class 113
15.7 ColumnSet Class Diagram 113
15.7.1 ColumnSet 114
15.7.2 Table Class 114
15.7.3 View Class 114
15.8 RecordData Class Diagram 115
15.8.1 RecordFile Class 115
15.9 XMLData Class Diagram 116
15.9.1 XMLSchema 116
15.10 XMLElements Class Diagram 116

15.10.1 XMLSimpleType 117
15.10.2 XMLRestriction 117
15.10.3 XMLComplexType 118
15.10.4 XMLAll Class 118
15.10.5 XMLSeq Class 118
15.10.6 XMLChoice Class 118
15.10.7 XMLOccurs Class 119
15.10.8 XMLGroup Class 119
15.10.9 XMLAny Class 119

15.11 ProgramElements Class Diagram 119

15.11.1 StoredProcedure Class 120
15.11.2 Query Class 120
15.11.3 DBTrigger Class 120

15.12 Key Relations class diagram 121

15.12.1 UniqueKey Class (additional properties) 121
15.12.2 ReferenceKey Class (additional properties) 121
15.12.3 KeyRelationship Class 122

16 Structure Package 123

16.1 Overview 123

16.2 Organization of the Structure Package 123

16.3 StructureInheritances Class Diagram 123

16.4 StructureModel Class Diagram 124

16.4.1 StructureModel Class 125
16.4.2 StructureElement Class 125
16.4.3 StructureGroup Class 125
16.4.4 StructureContainer Class 126
16.4.5 Subsystem Class 126
16.4.6 Layer Class 126
16.4.7 Component Class 126
16.4.8 SoftwareSystem Class 127
16.4.9 CodeResource (additional properties) 127

17 Event Package 129

17.1 Overview 129

17.2 Organization of the Event Package 129

17.3 EventInheritances Class Diagram 129

17.4 EventModel Class Diagram 129

17.4.1 EventModel Class 130
17.4.2 EventElement Class (abstract) 130
17.4.3 EventGroup Class 131
17.4.4 EventContainer Class 131

17.5 EventElements Class Diagram 132
17.5.1 Trigger Class 132
17.5.2 Message Class 132
17.5.3 UIEvent Class 132
17.6 EventRelations Class Diagram 133
17.6.1 EventRelationship Class (abstract) 133
17.6.2 EventElement Class (additional properties) 134
17.6.3 Triggers Class 134
17.6.4 CallableElement (additional properties) 134
18 UI Package 135
18.1 Overview 135
18.2 Organization of the UI Package 135
18.3 UIInheritances Class Diagram 135
18.4 UIModel Class Diagram 136
18.4.1 UIModel Class 136
18.4.2 UIElement Class 136
18.4.3 UIContainer Class 137
18.5 Display Class Diagram 137
18.5.1 Display Class 138
18.5.2 Screen Class 138
18.5.3 Report Class 138
18.6 DisplayUnits Class Diagram 139
18.6.1 DisplayUnit Class 139
18.6.2 FixedDisplayUnit Class 140
18.6.3 VariableDisplayUnit Class 140
18.6.4 UIRelationship Class (abstract) 140
18.6.5 DisplaysData Class 140
18.6.6 UsesImage Class 141
18.6.7 StorableElement Class (additional properties) 141
18.6.8 Image Class (additional properties) 141
18.7 UIRelations Class Diagram 142
18.7.1 UIFlow Class 142
18.7.2 Renders Class 143
18.7.3 Displays Class 143
18.7.4 Display Class (additional properties) 143
18.7.5 UIElement Class (additional properties) 144
18.7.6 CallableElement (additional properties) 144
18.7.7 UIElement Class (additional properties) 144
18.8 UILayout Class Diagram 144
18.8.1 UsesLayout Class 145
18.8.2 Display Class (additional properties) 145
18.8.3 UIContainer Class (additional properties) 146

19 Platform Package 147

19.1 Overview 147

19.2 Organization of the Platform Package 147

19.3 PlatformInheritances Class Diagram 147

19.4 PlatformModel Class Diagram 148

19.4.1 PlatformModel Class 149
19.4.2 PlatformElement Class (abstract) 149
19.4.3 PlatformGroup Class 150
19.4.4 PlatformContainer Class (abstract) 150

19.5 PlatformResources Class Diagram 151

19.5.1 PlatformPart Class 151
19.5.2 ResourceElement Class 152
19.5.3 ResourceInstance Class 152
19.5.4 ResourceType Class 153
19.5.5 ResourceDefinition Class 153

19.6 ResourceTypes Class Diagram 154

19.6.1 NamingResource Class 154
19.6.2 MarshaledResource Class 154
19.6.3 MessagingResource Class 155
19.6.4 DataResource Class 155
19.6.5 ExecutionResource Class 155
19.6.6 DataPortResource Class 155
19.6.7 DynamicData Class 156
19.6.8 DataManager Class 156

19.7 ExternalActors Class Diagram 156

19.7.1 ExternalActor Class 157
19.7.2 ExternalRelations Class (abstract) 157
19.7.3 Uses Class 157
19.7.4 ResourceInstance (additional properties) 158

19.8 PlatformInterfaces Class Diagram 158

19.8.1 CompliesTo Class 158
19.8.2 ResourceInstance (additional properties) 159
19.8.3 Interface (additional properties) 159

19.9 PlatformRelations Class Diagram 159

19.9.1 BindsTo Class 160
19.9.2 ResourceInstance (additional properties) 161
19.9.3 CodeResource (additional properties) 161

19.10 ProvisioningRelations Class Diagram 161

19.10.1 PlatformProvider Class 162
19.10.2 TechnologyRelationship Class (abstract) 162
19.10.3 Requires Class 163
19.10.4 Provides Class 163

19.10.5 PlatformElement (additional properties) ... 164
19.10.6 DeployedComponent (additional properties) ... 164

19.11 PlatformActions Class Diagram ... 164

19.11.1 PlatformService Class ... 165
19.11.2 MarshaledCall Class ... 166
19.11.3 AsynchCall Class ... 166
19.11.4 Registration Class ... 166
19.11.5 Activation Class ... 166
19.11.6 ResourceRelationship Class (abstract) ... 167
19.11.7 ResourceInstance (additional properties) ... 167

20 Runtime Package ... 169

20.1 Overview ... 169

20.2 Organization of the Runtime Package ... 170

20.3 RuntimeInheritances Class Diagram ... 171

20.4 RuntimeModel Class Diagram ... 171

20.4.1 RuntimeModel Class ... 172
20.4.2 RuntimeElement Class (abstract) ... 172
20.4.3 RuntimeContainer Class (abstract) ... 173
20.4.4 RuntimeGroup Class ... 173

20.5 Runnable Class Diagram ... 174

20.5.1 RunnableElement (abstract) ... 174
20.5.2 Process Class ... 174
20.5.3 Thread Class ... 175

20.6 Deployment Class Diagram ... 175

20.6.1 DeployedComponent Class ... 176
20.6.2 DeployedSoftwareSystem Class ... 177
20.6.3 Machine Class ... 177
20.6.4 DeployedResource Class ... 178

20.7 RuntimeActions Class Diagram ... 178

20.7.1 RuntimeRelation Class (abstract) ... 179
20.7.2 RuntimeService Class ... 179
20.7.3 LoadingService Class ... 180
20.7.4 Loads Class ... 180
20.7.5 SpawningService Class ... 180
20.7.6 Spawns Class ... 181
20.7.7 DeployedComponent (additional properties) ... 181
20.7.8 RunnableElement (additional properties) ... 181

21 Conceptual Package ... 183

21.1 Overview ... 183

21.2 Organization of the Conceptual Package ... 184

21.3 ConceptualInheritances Class Diagram 184
21.4 ConceptualModel Class Diagram 185
21.4.1 ConceptualModel 185
21.4.2 ConceptualElement (abstract) 186
21.4.3 ConceptualContainer 186
21.4.4 ConceptualGroup 187
21.4.5 TermUnit 187
21.4.6 FactUnit 187
22 Behavior Package 189
22.1 Overview 189
22.2 Organization of the Behavior Package 190
22.3 BehaviorInheritances Class Diagram 190
22.4 BehaviorModel Class Diagram 191
22.4.1 BehaviorModel 192
22.4.2 BehaviorElement (abstract) 192
22.4.3 BehaviorGroup 192
22.4.4 BehaviorContainer 193
22.4.5 BehaviorUnit 193
22.4.6 ScenarioUnit 193
22.4.7 RuleUnit 194

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

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

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

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

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

OMG Middleware Specifications

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

Platform Specific Model and Interface Specifications

- CORBAservices

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

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

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

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

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

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

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

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

Helvetica/Arial - 10 pt: Exceptions

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

Issues

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

1 Scope

This specification defines a meta-model for representing information related to existing software assets and their operational environments (referred to as the Knowledge Discovery Meta-model (KDM)). This is the first in the series of specifications related to Architecture-Driven Modernization (ADM) activity. ADM facilitates modernization projects by insuring interoperability for exchange of data between tools provided by different vendors.

One common characteristic of various tools that address the ADM challenge is that they analyze the existing software assets (for example, source code modules, database descriptions, build scripts, etc.) to obtain existing systems knowledge. Each tool implements 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 shall provide a common repository structure that will facilitate 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 promote a common interchange format that will allow interoperability between existing modernization tools, services and their respective models.

2 Conformance

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 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 application involves multiple platform technologies and programming languages. In order to achieve interoperability and especially the integration of information about different facets of an Enterprise application from multiple analysis tools, this specification defines several of 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). A KDM domain consists of a collection of tightly-coupled KDM packages that provide users with the capability to represent aspects of the system under study according to a particular paradigm or formalism. 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 and Platform.

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.

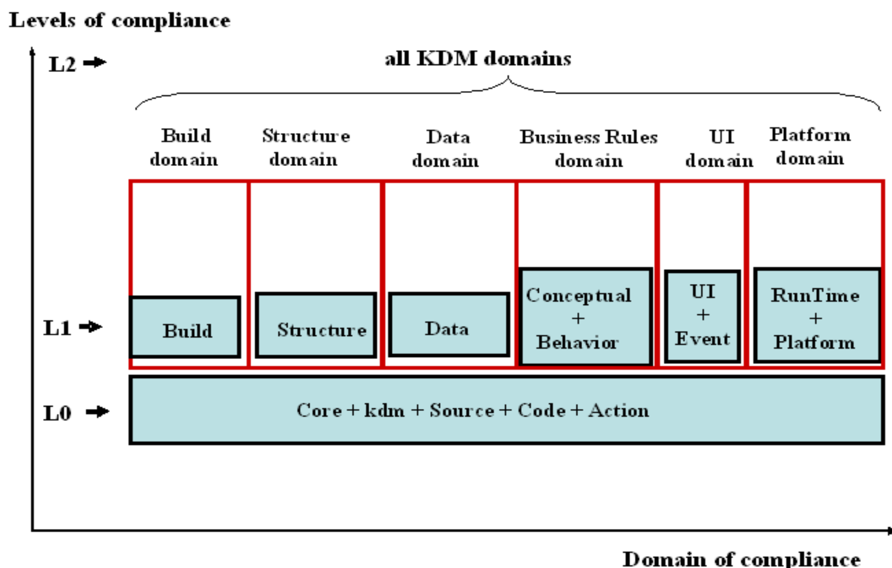

Figure 2.1 - Domains and levels of KDM compliance

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, Behavior, Conceptual, UI, Event, Runtime and Platform. 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 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 particular task.

  To be L1 compliant for a given KDM domain, a tool must completely support all model elements within all packages for that domain.

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

Package System allows integration of KDM models representing separate facets of knowledge of the same application. Therefore, package System is part of L0 compliance.

2.2.1 Meaning and Types of Compliance

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

For a given compliance level, there are:

- the capability to analyze physical artifacts of existing applications and output models based on the XMI schema corresponding to that compliance level.
- the capability to import models based on the XMI schema corresponding to that 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 elements of KDM framework; extend operations of existing tool to support 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 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.

Table 2.1 - Compliance Statements

	DATA	L0 compliance for analysis; extend operations of existing tool to support 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 elements of the Platform and Runtime packages.	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. Provide capability to analyze Runtime artifacts for specified platform; generate KDM Runtime model according to Runtime package.
	BUILD	L0 compliance for analysis; extend operations of existing tool to support 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.
	UI	L0 compliance analysis; extend operations of existing tool to support elements of the UI and Event packages.	L0 compliance for import; Support KDM API as defined by the UI and Event 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; generate KDM Event model according to Event package.
	BUSINESS	L0 compliance for analysis; extend operations of existing tool to support elements of the Conceptual and Behavior packages.	L0 compliance for import; Support KDM API as defined by the Conceptual and Behavior packages.	L0 compliance for export; Provide capability to analyze conceptual artifacts (e.g., domain concepts) of existing application and generate KDM Conceptual model according to Conceptual package; Provide capability to analyze behavior artifacts (e.g., business rules, scenarios) of existing application and generate KDM Behavior model according to Behavior package.
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 specification. For dated references, subsequent amendments to, or revisions of any of these publications do not apply.

- UML 2. Infrastructure Specification
- MOF 2.0 Specification

4 Terms and Definitions

There are no special terms or definitions in this specification.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Changes to Other OMG Specifications

There are no changes to other OMG specifications.

6.2 How to Read this Specification

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

Chapter 7. Specification overview

Provides design rationale for the KDM specification

Chapter 8. KDM

Gives the overview of the packages of KDM

Chapter 9. Core package

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

Chapter 10. kdm package

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.

Chapter 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.

Chapter 12. Code package

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

Chapter 13. Action package

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.

Chapter 14. Build package

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

Chapter 15. 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.

Chapter 16. 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.

Chapter 17. Event package

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

Chapter 18. UI package

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

Chapter 19. 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.

Chapter 20. Runtime package

Provides meta-model elements for representing the physical aspects of operating environments of software systems.

Chapter 21. Conceptual package

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

Chapter 22. Behavior package

Describes the meta-model elements for representing high-level behavior knowledge about existing software systems in the context of other KDM views.

6.3 Acknowledgements

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

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

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

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

7 Specification Overview

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

The KDM promotes a common interchange format that will allow interoperability between existing modernization tools, services, and their respective models. More specifically, (KDM) provides a common repository structure that facilitates the exchange of data currently contained within individual tool models that represent existing software assets. The meta-model represents the physical and logical software assets at various levels of abstraction as entities and relations.

KDM separates knowledge about existing systems into four orthogonal dimensions: Structure, Behavior, Data, and User Interface (UI) (refer to Figure 7.1). These dimensions represent well-known concerns in software engineering and correspond to Architecture Views.

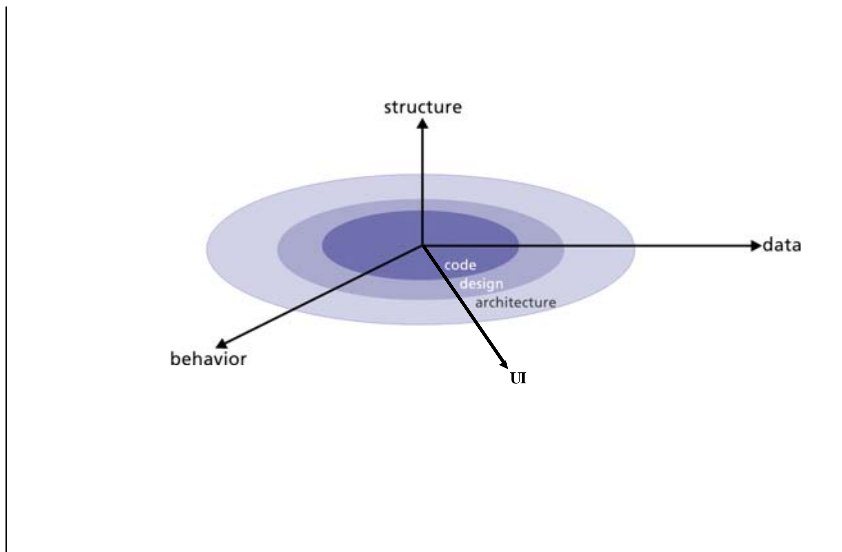

Figure 7.1 - Separation of concerns in software systems

Each of these dimensions 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-dependant 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 MOF model
- Common KDM defines additional semantic layer for interoperability on top of basic entity and relationships
- KDM can be extended to capture language-specific and application-specific entities and relationships
- Defines multiple hierarchies of entities via containers
- Defines composition of relationships
- For each entity there is a single predefined (home) hierarchy
- There are no arbitrary associations
- 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 derived relationships)
- Models are actionable
- Analyst is able to refactor the model (for example, by moving entities between containers) and map changes in the model to changes in the software through links

8 KDM

8.1 Overview

KDM specifies the core concepts required for understanding existing software in preparation for its modernization and provides infrastructure to support more detailed definitions of Knowledge Discovery.

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

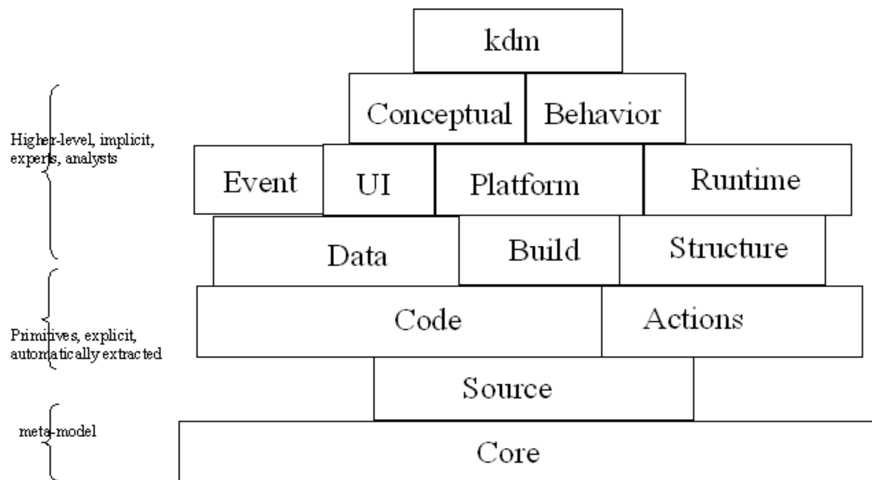

Figure 8.1 - Structure of KDM Packages

The KDM contains 14 packages; each package is defined by one or more class diagrams that refer to classes from other packages.

Core KDM package defines the basic meta-classes (entity, relationship, container hierarchies, etc.) and well-formedness rules of KDM models, including the lightweight extension mechanism.

8.2 Organization of the KDM Packages

The KDM is a collection of classes and associations that are described together because they provide meta-model constructs for defining existing software artifacts as entities and relations.

The KDM package has the following organization:

- The Core package defines the basic abstractions of KDM
- The Kdm package provides static context shared by all KDM models
- The Source package defines references to the source code
- The Code package defines the low-level building blocks of application source files, such as procedures, datatypes, classes, etc. (as determined by a programming language)
- Action package defines end points of relations, and the majority of KDM relations

- Build package defines the artifacts related to engineering of an application
- Data package defines the persistent data aspects of an application
- Structure package defines the architectural components of existing application, subsystems, layers, packages, etc.
- Event package defines a common concept related to event-based programming
- UI package defines the user-interface aspects of the application
- Platform package defines artifacts, related to the run time platform of the enterprise application
- Runtime package defines artifacts that are related to the run-time of an application on the target platform
- Conceptual package defines the domain-specific elements of an application
- Behavior package defines the scenario model

9 Core Package

9.1 Overview

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

9.2 Organization of the Core Package

The KDM uses packages to control complexity and create groupings of logically interrelated classes. The Core package is a collection of classes and associations that are described together because they provide meta-model constructs for other KDM packages.

The KDM Core classes are further grouped into the following 6 class diagrams:

- CoreEntities - a class diagram that represents key meta-model elements
- CoreRelations - a class diagram that represents key meta-model associations
- AggregatedRelations - a class diagram that represents the key analysis mechanism of KDM - the so-called aggregated relations
- Datatypes - a class diagram that defines utility data types
- Extensions - a class diagram that defines light-weight extension mechanism of KDM
- Annotations - a class diagram that provides user-defined attributes and annotations to the modeling elements

The Core package depends on no other packages.

9.3 CoreEntities Class Diagram

The Core class diagram defines key meta-model elements of KDM models.

The classes and associations that make up the CoreEntities class diagram are shown in Figure 9.1.

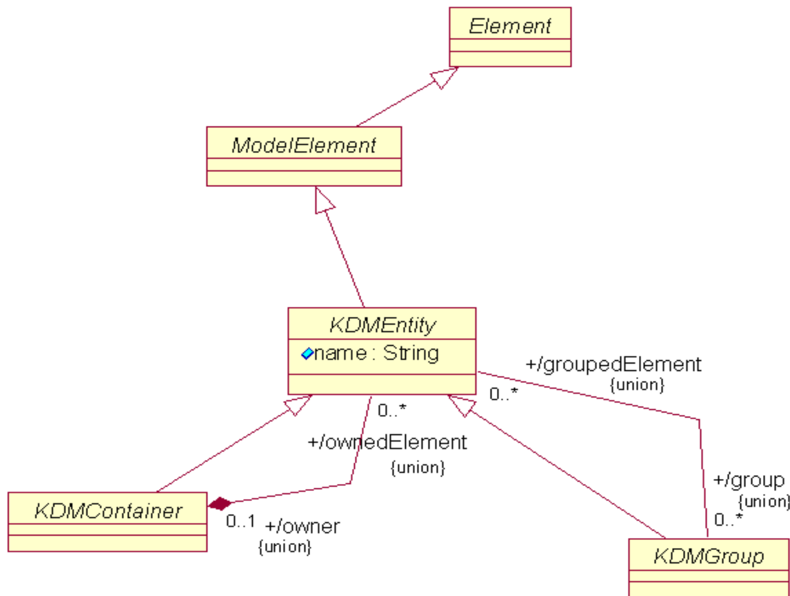

Figure 9.1 - CoreEntities Class Diagram

9.3.1 Element Class (Abstract)

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

Semantics

9.3.2 ModelElement Class (Abstract)

A model element is an element that is an abstraction drawn from the system being modeled.

In the meta-model, a ModelElement is the base for all modeling metaclasses in the KDM. All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

A ModelElement can be extended through the lightweight extension mechanism.

Superclass

Element

Semantics

9.3.3 KDMEntity Class (Abstract)

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

In the meta-model, KDMEntity is a subclass of ModelElement. Each KDM package defines some specific KDM entities that are direct or indirect subclasses of KDMEntity. Also, 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

Attributes

name: Name	An identifier for the KDM entity.

Associations

owner:KDMContainer[0..1]	KDM container to which the current element belongs. This property determines a meta-level interface to KDM entities. This property is a derived union. Every specific KDM container defines 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:KDMGroup[0..*]	KDM groups with which the current element is associated. This property determines a meta-level interface to KDM entities. This property is a derived union. Every specific KDM group defines 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

Semantics

9.3.4 KDMContainer Class (Abstract)

The KDMContainer is a part of a model that contains a set of KDMEntities.

In the meta-model, a KDMContainer is a KDMEntity that can own other KDMEntities. Each KDM entity can be owned by exactly one KDM container.

Superclass

KDMEntity

Associations

ownedElement : KDMEntity[0..*]	Entities that are associated with the container. This property determines a meta-level interface to KDM containers. This property is a derived union. Every specific KDM container defines 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.

Constraints

KDMEntity should be owned by exactly one KDMContainer.

Semantics

9.3.5 KDMGroup Class (Abstract)

The KDMGroup is a part of a model that contains a set of KDMEntities.

In the meta-model, a KDMGroup is a KDMEntity that can be associated with other KDMEntities. Each KDM entity can be associated with multiple KDM groups.

Superclass

KDMEntity

Associations

groupElement: KDMEntity[0..*]	KDM entities associated with the given group. This property determines a meta-level interface to KDM groups. This property is a derived union. Every specific KDM group defines 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.

Constraints

Semantics

9.4 CoreRelations Class Diagram

The Core class diagram defines key meta-model associations of KDM models.

The classes and associations that make up the CoreRelations class diagram are shown in Figure 9.2.

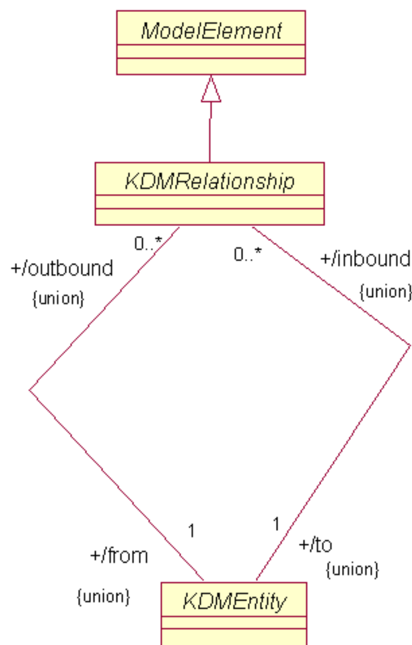

Figure 9.2 - CoreRelations Class Diagram

9.4.1 KDMRelationship Class (Abstract)

A KDM relationship is a model element that represents primitive 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. Also, each KDM package defines some specific KDM entities that are direct or indirect subclasses of KDMEntity. Specific subclasses of KDMRelationship are typed associations between some specific subclasses of KDMEntity.

Superclass

ModelElement

Associations

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

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

Constraints

To- and from-endpoints should be distinct.

Semantics

9.4.2 KDMEntity (additional properties)

Associations

inbound: KDMRelationship[0..*]	Primitive KDM relationships for which the given entity is the target. This property is an opposite of the to-endpoint of a KDMRelationship. This property determines a meta-level interface to KDM entities. This property is a derived union. Every specific KDM relationship redefines the from-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF "derived union" mechanism. Each concrete property subsets the "union" inbound property of the parent class, defined in the Core package.
outbound:KDMRelationship[0..*]	Primitive KDM relationships for which the given entity is the source. This property is an opposite of the from-endpoint of a KDMRelationship. This property determines a meta-level interface to KDM entities. This property is a derived union. Every specific KDM relationship redefines the to-endpoint to a particular subtype of KDMEntity. In KDM this is represented by the CMOF "derived union" mechanism. Each concrete property subsets the "union" inbound property of the parent class, defined in the Core package.

Semantics

9.5 AggregateRelations Class Diagram

The AggregateRelations class diagram defines the key analysis mechanism of KDM. AggregatedRelationship are part of the "meta-level" interface to KDM models, along with interface defined by KDMContainer, KDMGroup, KDMEntity, and KDMRelationship.

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

The classes that make up the AggregateRelations class diagram are shown in Figure 9.3.

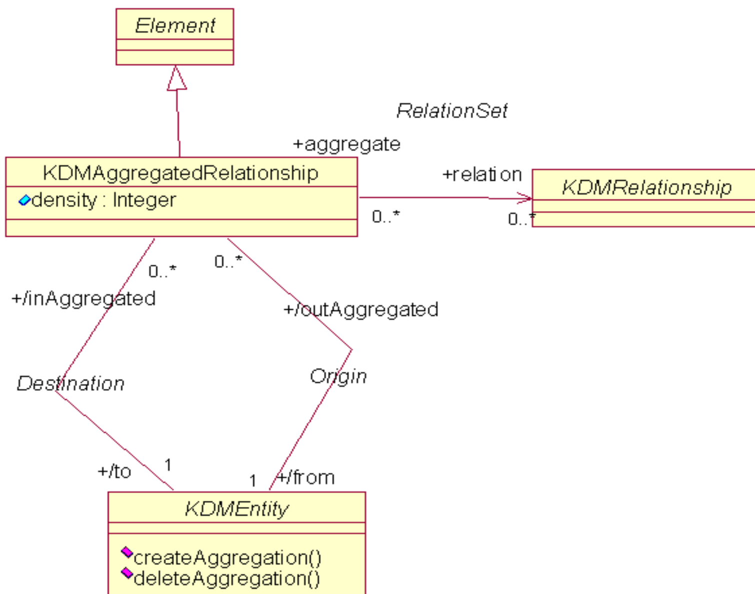

Figure 9.3 - AggregatedRelations Class Diagram

9.5.1 KDMAggregatedRelationship Class

The KDMAggregatedRelationship represents the set of aggregated relationships of the given entity. The set of derived relationships consists of the primitive relationships of the current entity and of all primitive relationships of the entities that are recursively contained in the current entity. 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). Overall management and lifecycle of the Aggregated Relationships is determined by the operations of the KDMEntity class.

Superclass

Element

Attributes

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

Associations

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

from:KDMEntity[1]	The source container of the relationships in the aggregated set. All relationships in the aggregated set should originate from the source container or from some entity that is contained directly or indirectly in the source container.

Operations

createAggregation(otherEntity:KDMEntity)	This operation creates an aggregated relationship such that the current entity is the from-endpoint of the aggregated relation and the "otherEntity" is the to-endpoint. The new aggregated relationship is owned by the model to which owns the current entity (either directly or indirectly through container ownership).
deleteAggregation (aggregatedRelation:KDMAggregatedRelationship)	This operation deletes the given aggregated relationship.

Constraints

To- and from-endpoints should be distinct.

The density should be greater than or equal to 1.

The density should be the same as the number of primitive relations represented by the given aggregated relationship.

The to- and from- endpoints of each relation should be the same as in the aggregated relationship.

Semantics

AggregatedRelations are determined by the grouping of elements into containers 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 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 and an entity represents the set of all regular relationships such that the to-endpoint is the given entity and the from-endpoint is any entity that is owned by the given container (directly or indirectly).

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

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

9.5.2 KDMEntity (additional properties)

Associations

inAggregated:KDMAggregatedRelationship[0..*]	Aggregated KDM relationships for which the given entity is the target.
outAggregated:KDMAggregatedRelationship[0..*]	Aggregated KDM relationships for which the given entity is the source.

Semantics

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 that make up the Datatypes class diagram are shown in Figure 9.4.

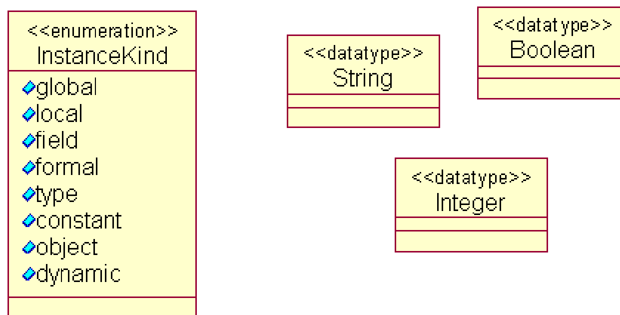

Figure 9.4 - Datatypes Class Diagram

9.6.1 InstanceKind Data Type (enumeration)

The meta-model InstanceKind defines an enumeration that clarifies the role of a data type. The values are defined below. The InstanceKind is used by TypeElement in Code package.

Literal Values

global	The data unit represents a global variable.
local	The data unit represents a local variable.
field	The data unit represents a field in a composite type or a class.
formal	The data unit represent a formal parameter.
type	The data unit represents a type.

constant	The data unit represents a constant.
object	The data unit represents an instance of a class.
dynamic	The data unit represents a dynamically created instance.

Semantics

9.6.2 Boolean Type (datatype)

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

Semantics

9.6.3 String Type (datatype)

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

Semantics

9.6.4 Integer Type (datatype)

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

Semantics

9.7 Extensions Class Diagram

The Extensions class diagram collects together classes and associations of the Core package that define the lightweight extension mechanism of KDM.

The classes and associations that make up the Extensions diagram are shown in Figure 9.5.

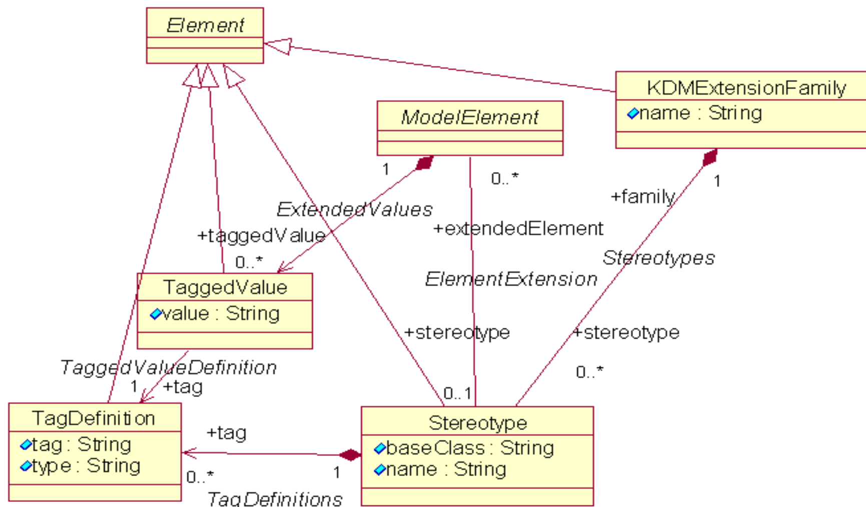

Figure 9.5 - Extensions Class Diagram

9.7.1 Stereotype Class

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

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

A Stereotype keeps track of the base classes to which it may be applied.

Superclass

Element

Attributes

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

Associations

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

extendedElement:ModelElement[1]	Designates the model element branded by the stereotype. Must be a model element of the kind specified by the baseClass attribute

Constraints

Tags associated with model element should not clash with any meta attributes associated with this model element.

A model element should have at most one tagged value with a given tag name.

A stereotype should not extend itself.

Semantics

9.7.2 TaggedDefinition Class

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

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

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

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

Superclass

Element

Attributes

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

Constraints

Semantics

9.7.3 ExtensionFamily Class

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

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

Superclass

Element

Attributes

name:String	Provides the name of the extension family.

Associations

tag:TagDefinition[0..*]	Stereotype owns the set of tag definitions that determine the additional tagged values associated with the model elements that are branded with the given stereotype.
extendedElement:ModelElement[1]	Designates the model element branded by the stereotype. Must be a model element of the kind specified by the baseClass attribute.

Constraints

Semantics

9.7.4 TaggedValue Class

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

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

Superclass

Element

Attributes

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

Associations

tag:TagDefinition[1]	The TagDefinition of this TaggedValue

Constraints

Semantics

9.7.5 ModelElement (additional properties)

Associations

taggedValue:TaggedValue[0..*]	The set of tagged values determined by the stereotype.
extension:Stereotype[0..1]	The stereotype

Constraints

Each tagged value added to a ModelElement must conform to a certain tag definition owned by the stereotype of that ModelElement (the tag association of the TaggedValue should refer to a TaggedDefinition that is owned by a Stereotype of the ModelElement). A tagged value conforms to a tag definition when the value conforms to the type. Conformance of lightweight extensions can only be validated dynamically, since lightweight extensions are not defined by the KDM standard.

Semantics

9.8 Annotations Class Diagram

The Annotations class diagram collects together classes and associations of the Core package that provide user-defined attributes and annotations to the modeling elements.

The classes and associations that make up the Annotations diagram are shown in Figure 9.6.

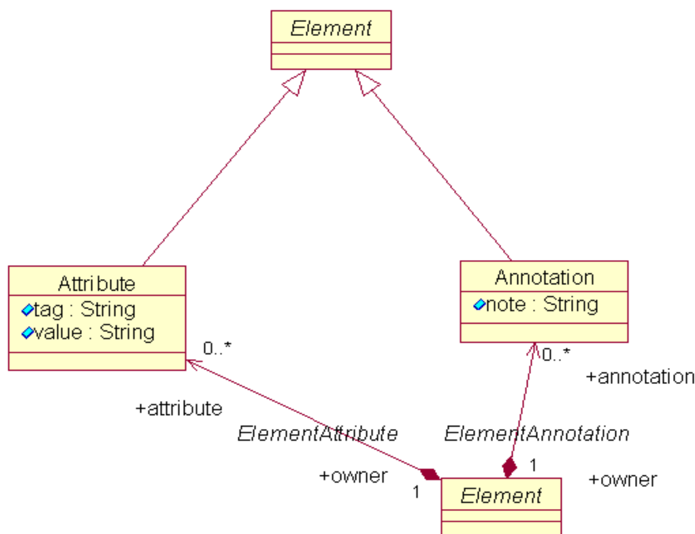

Figure 9.6 - Annotations Class Diagram

9.8.1 Attribute Class

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

The interpretation of attribute semantics is outside the scope of KDM. It must be determined by the user or tool conventions. It is expected that some tools will provide capability to add arbitrary attributes to the instances of the model to supply information needed for their operations beyond the basic semantics of KDM. Such information could support analysis of KDM models by analysis, etc.

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

Superclass

Element

Attributes

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

Constraints

Semantics

9.8.2 Annotation Class

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

Superclass

Element

Attributes

note:String	Contains the textual description of the target model element.

Constraints

Semantics

9.8.3 Element (additional properties)

Associations

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

Constraints

Semantics

10 KDM Package

10.1 Overview

The Kdm package defines key infrastructure elements that determine patterns for constructing KDM models, composition and integration of KDM models, and consequently the queries against KDM models.

10.2 Organization of the Kdm Package

The Kdm package is a collection of classes and associations that define the overall structure of KDM models. The key KDM artifact is referred to as "a KDM model." A KDM model is the minimal unit of exchange. A KDM model corresponds to one particular view of the existing system being modeled. A model 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 be 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 model segments that are exported by separate extractor tools, and may need to be integrated to provide a coherent holistic view. The integration tool will define the model Root that further refers to multiple segments.

Thus the KDMRoot element is an n entry point into a KDM model. Each model Segment includes one or more KDM models.

Root element may also contain one or more KDM models that are "global" (shared between multiple model segments). For example, all Platform, Runtime, Conceptual, and Behavior models can be associated with the Root element to avoid duplication in individual model segments.

Root, Segment, and individual KDM models are static element of KDM models that are not dependent of the software systems that are being modeled by KDM. In the meta-model these classes are subclasses of Element, rather than of ModelElement. They determine the overall infrastructure of KDM, as opposed to the model elements from the Core package that determine the structure of the existing application being modeled.

The Kdm package consists of the following class diagrams:

- Framework – defines the elements of the kdm framework.
- ModelRoot – defines the individual KDM models and how they are associated with the framework.
- Audit –defines audit information for KDM model elements.

The Kdm package depends on the following packages:

org.omg::ADM::KDM::Action
org.omg::ADM::KDM::Behavior
org.omg::ADM::KDM::Build
org.omg::ADM::KDM::Code
org.omg::ADM::KDM::Conceptual
org.omg::ADM::KDM::Data
org.omg::ADM::KDM::Event
org.omg::ADM::KDM::Platform

org.omg::ADM::KDM::Runtime
org.omg::ADM::KDM::Structure
org.omg::ADM::KDM::UI

10.3 Framework Class Diagram

The Framework class diagram defines the model Segment.

The classes and association of the Framework diagram are shown in Figure 10.1.

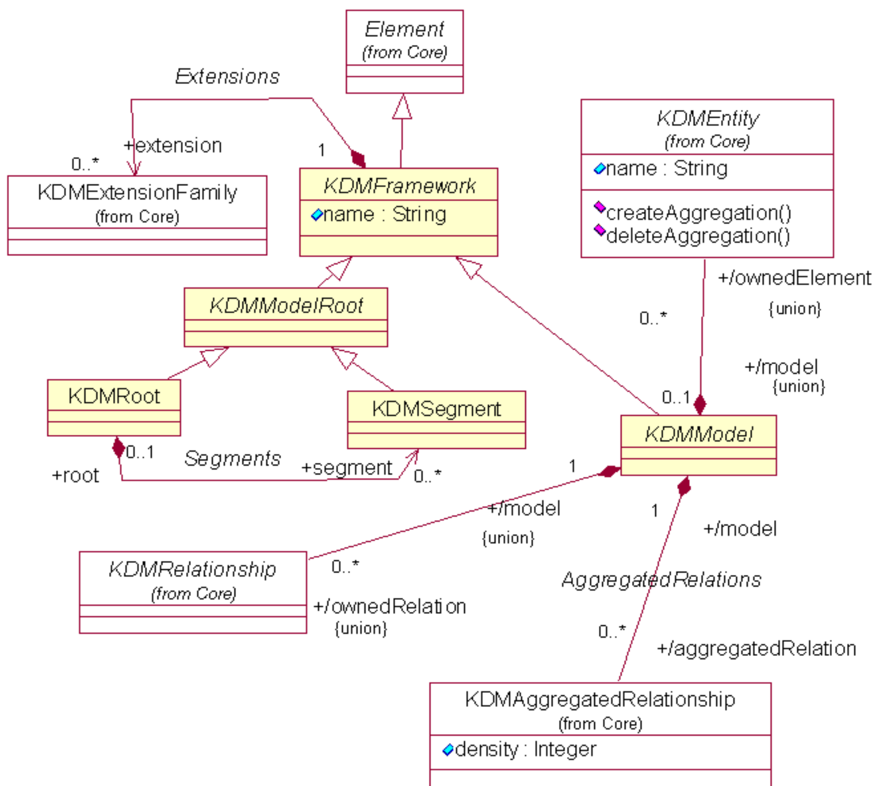

Figure 10.1 - Framework Class Diagram

10.3.1 KDMFramework Class (abstract)

The KDMFramework element is the common parent for the KDM framework classes.

Superclass

Element

Attributes

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

Associations

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

Constraints

Semantics

10.3.2 KDMModel Class (Abstract)

A KDM model represents a certain architecture view of software systems. Each KDM model completely describes certain aspects of software systems according to the separation of concerns principle. KDM model is a minimal unit of exchanging information between tools.

KDM defines several specific KDM models.

In the meta-model, KDMModel extends the Element class. A KDM package defines a subclass of KDMModel and the corresponding specific subclasses of KDMEntity and KDMRelationship. Specific subclasses of KDMModel serve as typed containers for some specific subtypes of KDMEntity and KDMRelationship. KDMModel serves as the container for KDMAggregatedRelationships.

Superclass

KDMFramework

Associations

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

Constraints

Semantics

10.3.3 KDMSegment Class

The KDMSegment element represents a coherent set of related KDM models representing an existing software system. KDMSegment is a typical unit of exchange between tools.

Superclass

KDMModelRoot

10.3.4 KDMRoot Class

The KDMRoot element represents a coherent set of related KDM models representing an existing software system. KDMRoot is a typical unit of exchange between tools.

Superclass

KDMModelRoot

Associations

segment: KDMSegment[0..*]	Instances of KDM entities owned by the model. Each KDM model defines specific subclasses of KDMEntity class.

Constraints

Semantics

10.4 ModelRoot Class Diagram

The ModelRoot class diagram defines the ModelRoot element.

The classes and associations of the ModelRoot diagram are shown in Figure 10.2.

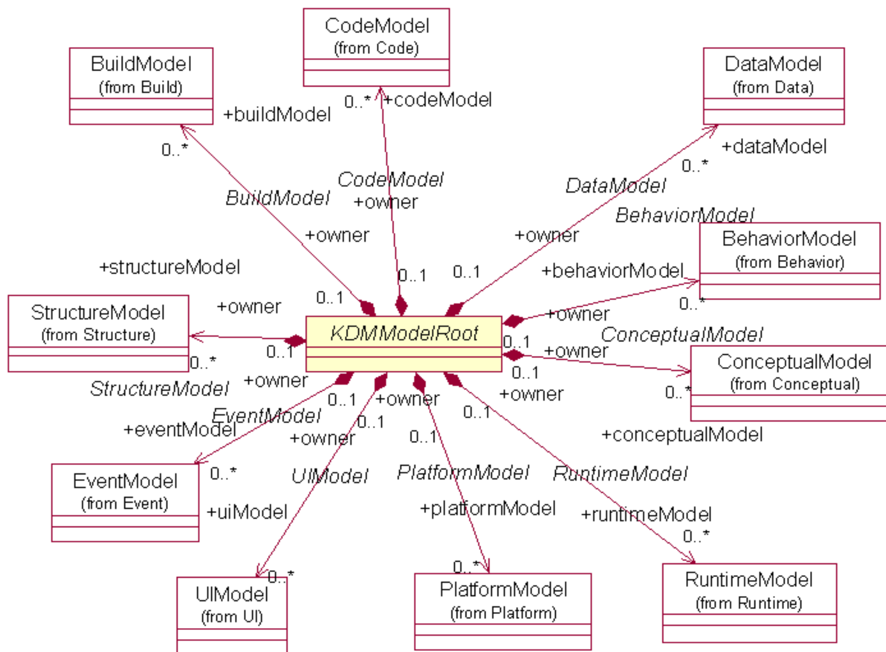

Figure 10.2 - ModelRoot Class Diagram

10.4.1 ModelRoot Class (abstract)

ModelRoot element is the entry point into a KDM model. This is an infrastructure element that provides links to the "global" KDM models that are shared among multiple model segments, as well as to the set of segments of the model.

Superclass

KDMFramework

Associations

codeModel: CodeModel[0..*]	Represents basic program elements and their relations in the target Segment. For example, data types, procedures, variables, classes, methods.
dataModel: DataModel [0..*]	Represents basic elements of persistent storage and their relations in the target Segment.
platformModel:PlatformModel[0..*]	Represents platform elements used by individual model Segment. For example, platform-specific definitions of resources and corresponding relations.
runtimeModel:RuntimeModel[0..*]	Represents physical deployment of the software system as well as the dynamic structures and the corresponding relations.
conceptualModel:ConceptualModel[0..*]	Represents domain-specific entities and relations.
behaviorModel:BehaviorModel[0..*]	Represents domain specific rules and scenarios.
structureModel:StructureModel[0..*]	Represents architecture components that are shared by individual model Segments.
uiModel: UIModel [0..*]	Represents basic elements of the user interface and their relations.
eventModel: EventModel [0..*]	Represents basic events and their relations in the target Segment.
buildModel:BuildModel[0..*]	Represents basic elements of the build and their relations for the target Segment.

Constraints

Semantics

10.5 Audit Class Diagram

The Audit class diagram collects together classes and associations of the Core package that defines audit information for KDM model elements.

The classes and associations that make up the Audit class diagram are shown in Figure 10.3.

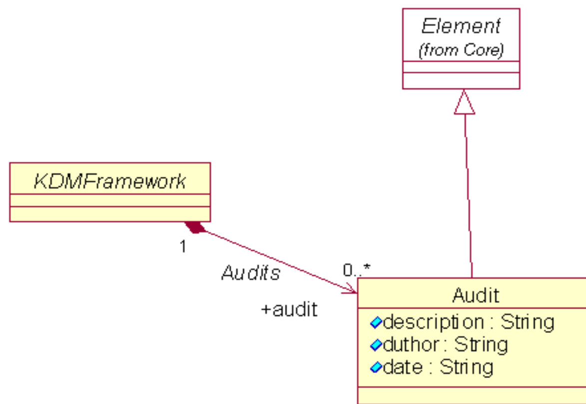

Figure 10.3 - Audit Class Diagram

10.5.1 Audit Class

Audit class represents basic audit information associated with KDM models.

Superclass

Element

Attributes

description:String	Contains the description of the model element.
author:String	Contains the name of the person who has created the model element, or the name of the tool that was used to create the model element.
date:String	Contains the date when the model element was created.

Constraints

Semantics

10.5.2 KDMFramework (additional properties)

Associations

audit:Audit[0..*]	This association links a KDMFramework element with the set of audit attributes.

Constraints

Semantics

11 Source Package

11.1 Overview

The Source package provides meta-model constructs for specifying the link between the model artifacts and their physical representation by the artifacts of the existing system. It represents the convergence between the model representation and the application source.

The Source package offers two capabilities for linking KDM model to the corresponding artifacts:

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

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

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

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

The nature of the “source code” that implements a particular KDM element is outside of the scope of KDM. In KDM, this is indicated by the “language” attribute.

11.2 Organization of the Source Package

The Source package consists of two diagrams:

- SourceRef – the link between a model element and the source code.
- SourceRegion – the link between a model element and the corresponding physical artifact.

The Source package depends on the following packages:

org.omg::ADM::KDM::Build

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Core

org.omg::ADM::KDM::Data

org.omg::ADM::KDM::UI

11.3 SourceRef Class Diagram

The SourceRef class diagram defines classes and associations that link elements of the KDM model of an existing software system to the physical artifacts of that system.

The class diagram shown in Figure 11.1 captures these classes and their relations.

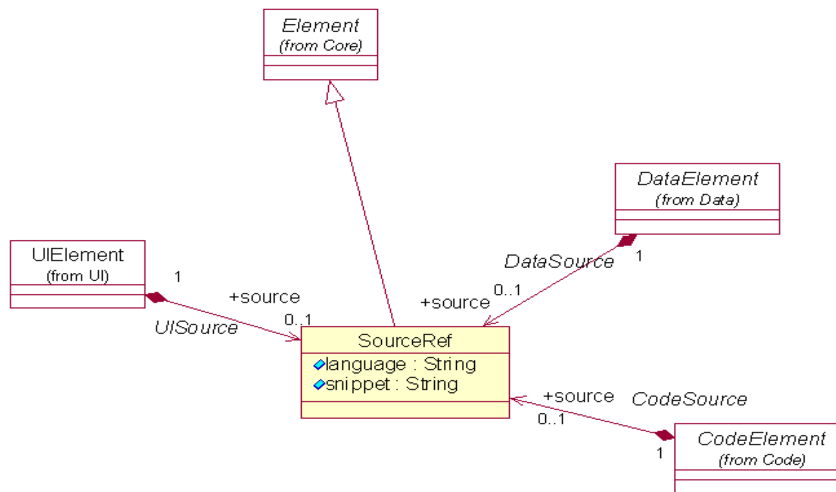

Figure 11.1 - SourceRef Class Diagram

11.3.1 SourceRef Class

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

Superclass

Element

Attributes

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

Constraints

Language indicator has to be provided using at least one of the following methods:

- As the attribute of the SourceRef element.
- As the attribute of the SourceRegion element.
- As the attribute of the SourceFile element (part of the Build Model), accessible via the SourceRegion element.

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

Semantics

11.3.2 CodeElement (additional properties)

This association links a particular CodeElement to its source code.

Associations

source: SourceRef[0..1]	Source for the given KDM element.

Semantics

11.3.3 DataElement (additional properties)

This association links a particular DataElement to its source code.

Associations

source: SourceRef[0..1]	Source for the given KDM element.

Semantics

11.3.4 UIElement (additional properties)

This association links a particular UIElement to its source code.

Associations

source: SourceRef[0..1]	Source for the given KDM element.

Semantics

11.4 SourceRegion Class Diagram

The SourceRegion class diagram defines classes and associations. that define links between KDM model elements and the physical artifacts of the existing system.

The class diagram shown in Figure 11.2 captures these classes and their relations.

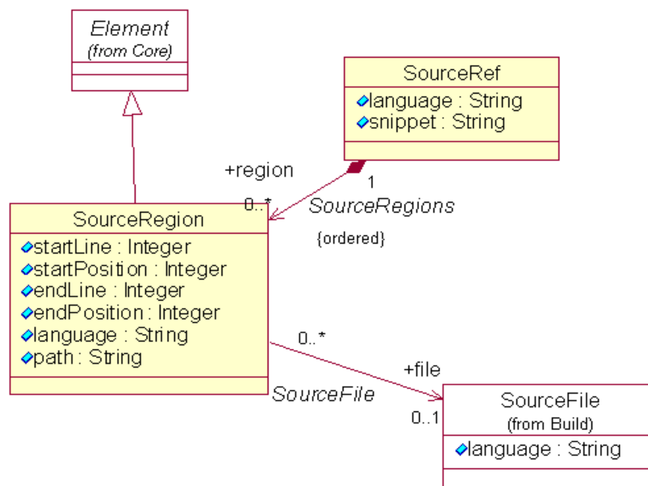

Figure 11.2 - SourceRegion Class Diagram

11.4.1 SourceRegion Class

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

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

Superclass

Element

Attributes

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

Associations

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

Constraints

The location of the source file should be provided using at least one of the following methods:

- Path attribute of the SourceRegion element.
- Path attribute of the SourceFile element of the Build model.

Semantics

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

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

12 Code Package

12.1 Overview

The Code package contains meta-model classes and associations capturing system artifacts to model the structure of programming languages and their relationships. It includes classes to model structural programming artifacts such as data types, data items, classes, procedures, macros, prototypes, and templates.

12.2 Organization of the Code Package

The Code package consists of the following class diagrams:

- CodeInheritances
- CodeModel
- CodeRelations
- CallableUnits
- Modules
- Prototype
- Macro
- Template
- TemplateRelations
- SimpleTypes
- PredefinedTypes
- DerivedTypes
- CompositeTypes
- ClassTypes
- EnumeratedTypes
- Signature
- Interface
- ClassRelations
- TypeRelations
- InterfaceRelations
- Comment
- Visibility

The Code package depends on the following packages:

org.omg::ADM::KDM::Action
org.omg::ADM::KDM::Source
org.omg::ADM::KDM::Core

12.3 CodeInheritances Class Diagram

The CodeInheritances class diagram define how classes of the Code package inherit from the Core package. The class diagram shown in Figure 12.1 captures these relations.

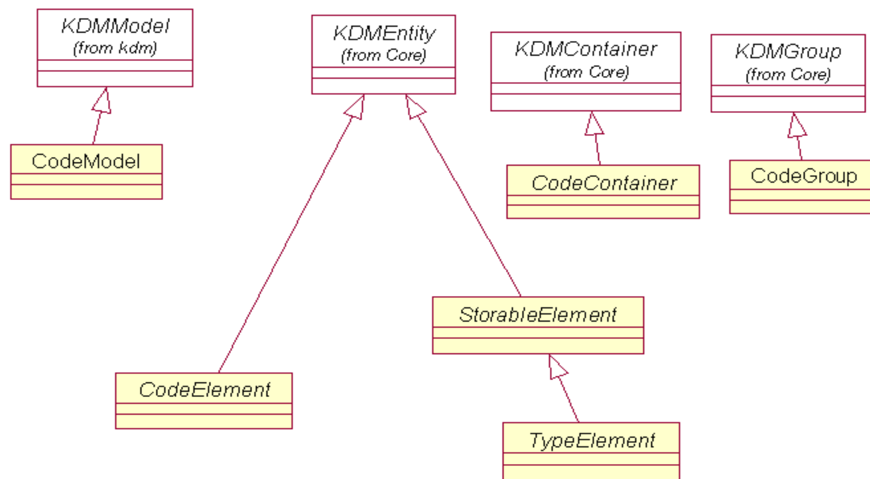

Figure 12.1 - CodeInheritances Class Diagram

12.4 CodeModel Class Diagram

The CodeModel class diagram collects together classes and associations of the Code package. They provide basic meta-model constructs to define a language-independent structure of code artifacts.

The CodeModel diagram describes the following types:

- CodeModel – a class representing a model for CodeElement.
- CodeElement – a class representing an abstract parent class for all KDM entities that can be used to model code.
- CodeResource – is the key abstract class that represents structural elements determined by a programming language
- CodeGroup – a class representing a grouping of CodeElement.
- CodeContainer – a class representing a container for CodeElement allowing composition and a hierarchical structure of CodeElement.
- TypeElement – a subclass of CodeResource that represents data items and type definitions.
- TypeContainer – a subclass of CodeContainer and TypeElement representing a container for TypeElement allowing composition and hierarchical structure of TypeElement.

The class diagram shown in Figure 12.2 captures these classes and their relations.

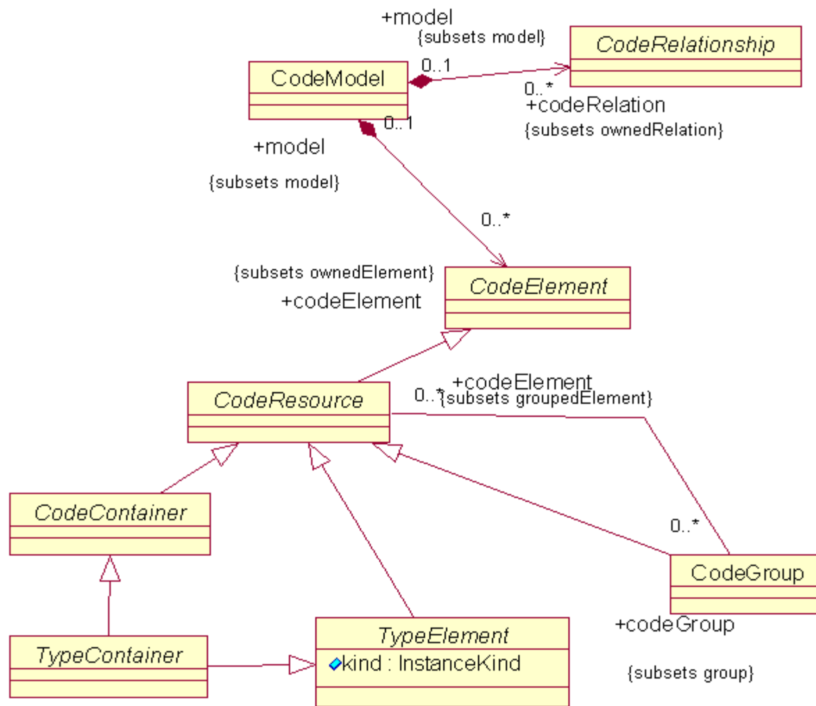

Figure 12.2 - CodeModel Class Diagram

12.4.1 CodeModel Class

The CodeModel is the specific KDM model for the Code package.

Superclass

KDMModel

Associations

codeElement:CodeElement[0..*]	The set of the top-level elements that are defined in this code model. The CodeModel element is the owner of such CodeElement. Some of these elements may be CodeContainers that own additional CodeElements. This property subsets the ownedElement property of KDMModel derived union.
codeRelation:CodeRelation[0..*]	The set of CodeRelations owned by this code model.

Constraints

Semantics

12.4.2 CodeElement Class (abstract)

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

Superclass

KDMEntity

Constraints

Semantics

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

12.4.3 CodeResource Class (abstract)

CodeResource class represents the named elements determined by the programming language (the so-called "symbols," "definitions," etc.). There are CodeElements that are not CodeResources, for example ActionElements that are defined in the Action package. CodeResource can be arranged in CodeGroup and organized in a hierarchy using CodeContainer.

Associations

codeGroup:CodeGroup[0..*]	The set of CodeGroup with which the current CodeResource is associated.

Semantics

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

12.4.4 CodeGroup Class

The CodeGroup is a general purpose grouping of CodeElement. This allows a CodeElement to be arranged in multiple CodeGroups. CodeGroup subclasses the KDMGroup and constraints the groupedElement property. Instead of any KDMElement, a CodeGroup can contain only CodeElement.

Superclass

CodeElement

KDMGroup

Associations

groupedElement:CodeElement[0..*]

Constraints

Semantics

CodeGroup is a concrete KDM modeling element.

12.4.5 CodeContainer Class (abstract)

The CodeContainer is a container for CodeElement. It allows composition and hierarchical structure. CodeContainer is further subclassed by more specific containers that subset the ownedElement property of the KDMContainer class.

Superclass

CodeElement

KDMContainer

Constraints

Semantics

CodeContainer is an abstract class that is used to define containers in the Code model with constrained owned elements.

12.4.6 TypeElement Class (abstract)

The TypeElement is a generic meta-model element that represents type entities of software systems. In the meta-model a TypeElement is a subclass of KDMEntity.

Superclass

CodeElement

KDMEntity

StorableElement

Attributes

kind : InstanceKind	The indicator of the kind of the data item represented by this element.

Constraints

Semantics

TypeElement is an abstract class that is used to constrain owned elements in KDM containers that represent complex types.

12.4.7 TypeContainer Class (abstract)

The TypeContainer is a meta-model element that provides generic grouping capabilities for TypeElements. TypeContainer owns a set of TypeElements. A TypeElement can be associated with a single TypeContainer.

In the meta-model TypeContainer is a subclass of KDMContainer. It is also a subclass of TypeElement. TypeContainer class is further sub-classed by several specific KDM type container classes.

Superclass

CodeContainer

TypeElement

Constraints

Semantics

TypeContainer is an abstract class that is used to represent KDM containers with restricted owned elements.

12.5 CodeRelations Class Diagram

The CodeRelations class diagram defines several abstract classes that represent various KDM relationship families. The class diagram shown in Figure 12.3 captures these classes and their relations.

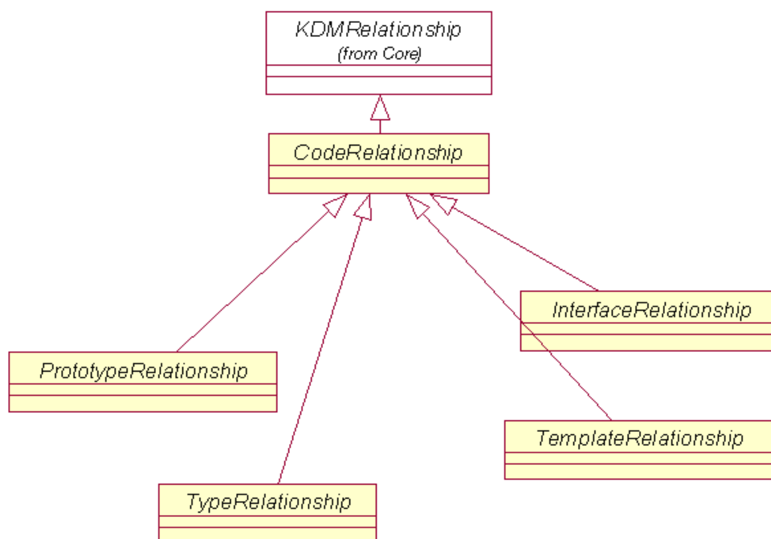

Figure 12.3 - CodeRelations Class Diagram

12.5.1 InterfaceRelationship Class (abstract)

The InterfaceRelationship class is an abstract class that represents the family of KDM relations describing the usages of interfaces.

Superclass

KDMRelationship

Semantics

12.5.2 TemplateRelationship Class

The TemplateRelationship class is an abstract class that represents the family of KDM relations describing the usages of templates.

Superclass

KDMRelationship

Semantics

12.5.3 TypeRelationship Class

The TypeRelationship class is an abstract class that represents the family of KDM relations describing the usages of types.

Superclass

KDMRelationship

Semantics

12.5.4 PrototypeRelationship Class

The PrototypeRelationship class is an abstract class that represents the family of KDM relations describing the usages of prototypes.

Superclass

KDMRelationship

Semantics

12.6 CallableUnits Class Diagram

The CallableUnits class diagram defines basic meta-model constructs to represent CallableElement, such as procedures, functions, methods, etc. The class diagram shown in Figure 12.4 captures these classes and their relations.

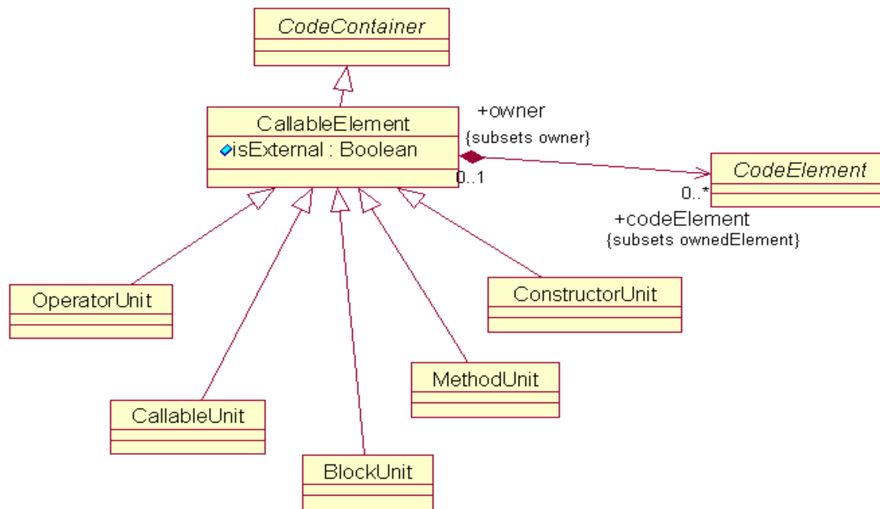

Figure 12.4 - CallableUnits Class Diagram

12.6.1 CallableElement Class

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

Superclass

CodeContainer

Attributes and Associations

isExternal:Boolean	Provides Boolean value to define if the element represents an external service.
codeElement:CodeElement[0..*]	Represents owned elements, such as local definitions and actions.

Constraints

Semantics

12.6.2 CallableUnit Class

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

Superclass

CodeElement

Semantics

12.6.3 BlockUnit Class

The BlockUnit represents a block of code for block-structured programming languages.

Superclass

CodeElement

Semantics

12.6.4 MethodUnit Class

The MethodUnit represents member functions owned by a ClassUnit.

Superclass

CodeElement

Semantics

12.6.5 ConstructorUnit Class

The ConstructorUnit represents the object oriented language concept constructor called when an object is created. The ConstructorUnit is owned by the ClassUnit.

Superclass

CallableElement

Semantics

12.6.6 OperatorUnit Class

The OperatorUnit is a CallableElement. It is used to model the object oriented language concept of a user-defined operator.

Superclass

CallableElement

Semantics

12.7 Module Class Diagram

The Module class diagram collects together classes and associations of the Code package that represent packaging aspects of programming languages, such as compilation units, shared files, and binary components. The class diagram shown in Figure 12.5 captures these classes and their relations.

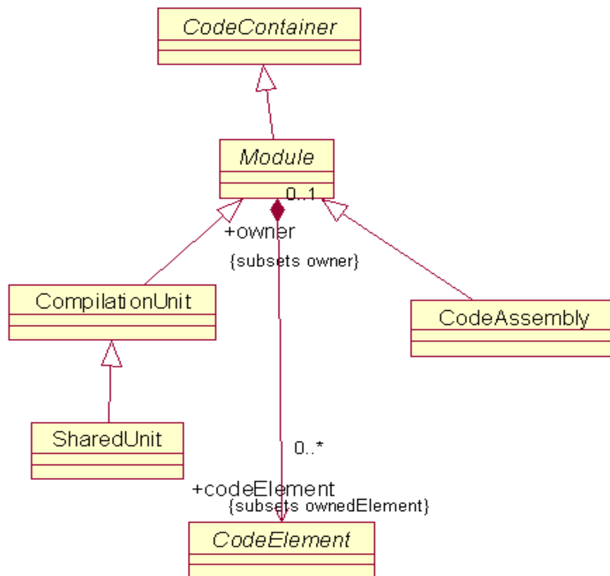

Figure 12.5 - Module Class Diagram

12.7.1 Module Class

The Module is a subtype for CodeContainer to model a software module or component, which either requires building or is used by the build process, and which can be allocated to deployed components. Module owns CodeElements and determines relations between them.

Superclass

CodeContainer

Associations

ownedElement:CodeElement[0..*]	The set of owned CodeElement.

Semantics

12.7.2 CompilationUnit Class

The CompilationUnit is a subtype for Module to model the source file as an owner of certain code elements, determined by the build process.

Superclass

Module

Semantics

12.7.3 SharedUnit Class

The SharedUnit is a subtype for Module to model the shared source files as determined by the build process.

Superclass

Module

Semantics

12.7.4 CodeAssembly Class

The CodeAssembly is a subtype for Module to model the binary object file resulting of a build process.

Superclass

Module

Semantics

12.8 Prototype Class Diagram

The Prototype class diagram provides basic meta-model constructs to define the relationship between PrototypeUnit and CodeElement. The class diagram shown in Figure 12.6 captures these classes and their relations.

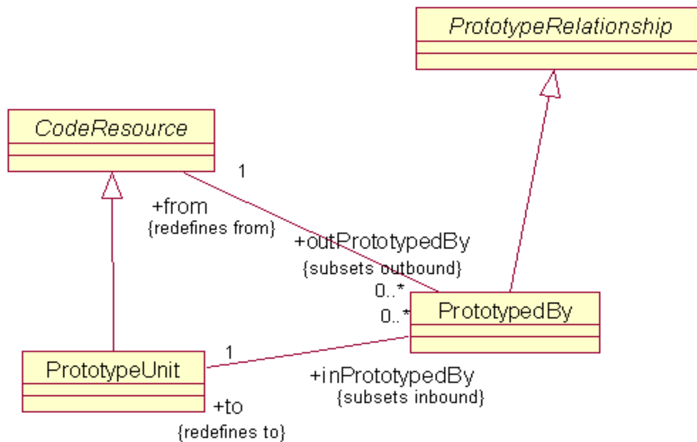

Figure 12.6 - Prototype Class Diagram

12.8.1 PrototypeUnit Class

The PrototypeUnit is a CodeElement and is used to model prototype constructs. Prototype constructs are declaration directives in programming languages used by compilers to verify argument lists and definition.

Superclass

CodeElement

Associations

inPrototypedBy:PrototypedBy[0..*]	The incoming PrototypedBy relations.

Constraints

Semantics

12.8.2 PrototypedBy Class

The PrototypedBy is an association class to provide linkages between CodeElement and PrototypeUnit.

Superclass

PrototypeRelationship

Associations

from:CodeElement[1]

to:PrototypeUnit[1]

Constraints

Semantics

12.8.3 CodeElement Class (additional properties)

Associations

outPrototypedBy:PrototypedBy[0..*]

Constraints

Semantics

12.9 Macro Class Diagram

The Macro class diagram provides basic meta-model constructs to define macroprocessing capabilities that are provided by certain programming languages. The class diagram shown in Figure 12.7 captures these classes and their relations.

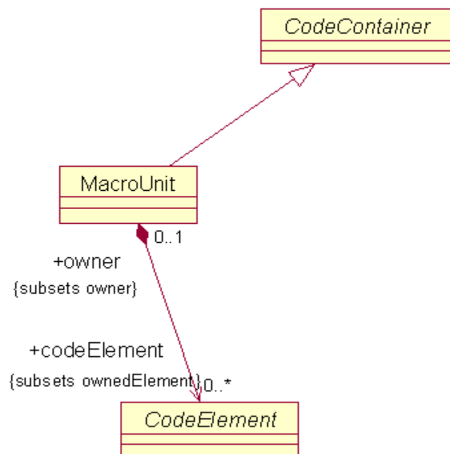

Figure 12.7 - Macro Class Diagram

12.9.1 MacroUnit Class

The MacroUnit is a CodeContainer. MacroUnit is used to model macro type constructs used by programming languages to define a sequence of code that is replaced and expanded during pre-processing of the code by the compiler or interpreter.

Superclass

CodeElement

Associations

codeElement:CodeElement[0..*]

Constraints

Semantics

12.10 Template Class Diagram

The Template class diagram provide basic meta-model constructs to define templates, parameters, instantiations of template and their relationships. The class diagram shown in Figure 12.8 captures these classes and their relations.

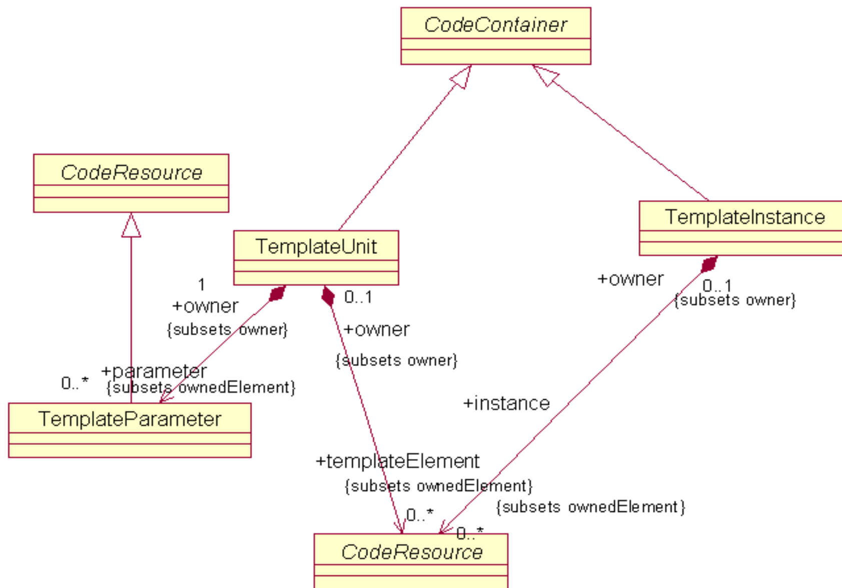

Figure 12.8 - Template Class Diagram

12.10.1 TemplateUnit Class

The TemplateUnit is a CodeContainer and allows composition for TemplateParameter and arbitrary CodeResource.

Superclass

CodeContainer

Associations

parameter:TemplateParameter[0..*]

TemplateElement:CodeResource[0..*]

Constraints

Semantics

12.10.2 TemplateParameter Class

The TemplateParameter is a CodeElement and is a parameter for TemplateUnit.

Superclass

CodeResource

Constraints

Semantics

12.10.3 TemplateInstance Class

The TemplateInstance is a CodeContainer representing the instantiation of a TemplateUnit. Template instance contains all instances resulting from the template instantiation because they are endpoints of the actual relations.

Superclass

CodeContainer

Associations

instance:CodeResource[0..*]

Constraints

Semantics

12.11 TemplateRelations Class Diagram

The TemplateRelations class diagram defines meta-model constructs to define KDM relationships between templates and instanciations of templates.

The TemplateRelations diagram describes the following types:

- Instantiates – an association class linking TemplateUnit to TemplateInstance.
- ProgramElement – an interface for TemplateInstance to enable instantiation.
- InstanceOf – an association class linking ProgramElement instances to TemplateInstance.

The class diagram shown in Figure 12.9 captures these classes and their relations.

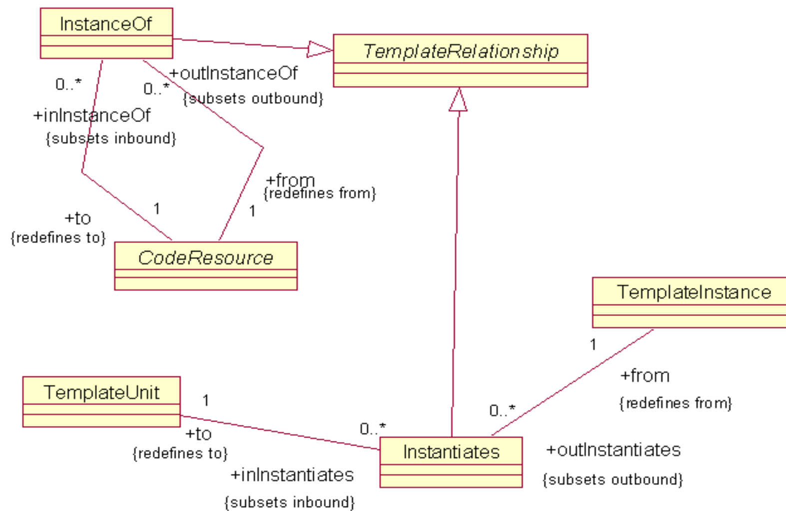

Figure 12.9 - TemplateRelations Class Diagram

12.11.1 Instantiates Class

The Instantiates is a subtype of TemplateRelationship and is used to provide linkages between TemplateUnit and TemplateInstances.

Superclass

TemplateRelationship

Associations

from:TemplateInstance[1]

to:TemplateUnitUnit[1]

Constraints

Semantics

12.11.2 InstanceOf Class

The InstanceOf represents a relationship between a CodeResource that is part of a TemplateInstance and the corresponding definition in the TemplateUnit.

Superclass

TemplateRelationship

Associations

from:CodeResource[1]

to:CodeResource[1]

Constraints

Semantics

12.11.3 CodeResource (additional properties)

Associations

outInstanceOf:InstanceOf[0..*]

inInstanceOf:InstanceOf[0..*]

Constraints

Semantics

12.12 SimpleTypes Class Diagram

The SimpleTypes class diagram defines meta-model element that represent simple data elements and data type elements as determined by programming languages. The classes and associations that make up the SimpleTypes diagram are shown in Figure 12.10.

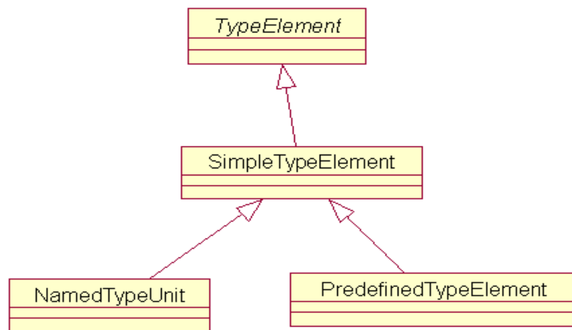

Figure 12.10 - SimpleTypes Class Diagram

12.12.1 SimpleTypeUnit Class

The SimpleTypeUnit is a meta-model element that represents an instance of a type. SimpleTypeUnit is a common superclass for PredefinedTypeElement that represents instances of built-in types determined by programming languages, and NamedTypeUnit, that represents user-defined types. Depending on the context, this can represent a variable definition, an element of a complex data type, such as a field in a record, a formal parameter of a signature, etc.

Superclass

TypeElement

Constraints

Semantics

12.12.2 NamedTypeUnit Class

The NamedTypeUnit is a meta-model element that represents an instance of a user-defined (named) type. Depending on the context, this can represent a variable definition, an element of a complex data type, such as a field in a record, a formal parameter of a signature, etc.

Superclass

SimpleTypeElement

Constraints

12.12.3 PredefinedTypeElement Class

The PredefinedTypeElement is a generic meta-model element that represents predefined data types provided by a programming language.

In the meta-model PredefinedTypeElement is a subclass of SimpleTypeElement, which allows it to be the endpoint of various type relationships. This class is an extension point. In KDM PredefinedTypeElement is subclassed by several specific classes.

Superclass

SimpleTypeElement

Semantics

12.13 PredefinedTypes Class Diagram

The PredefinedTypes class diagram defines model elements that represent predefined types common to various programming languages. The classes and association that make up the PredefinedTypes diagram are shown in Figure 12.11.

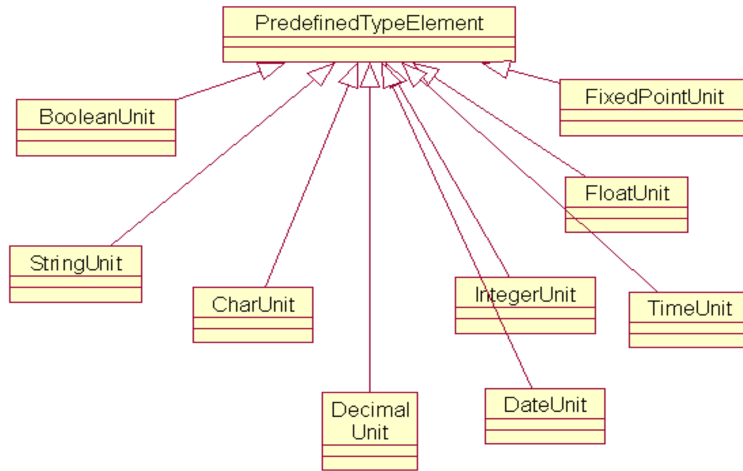

Figure 12.11 - PredefinedTyped Class Diagram

12.13.1 StringUnit Class

The StringUnit is a meta-model element that represents string data type common to various programming languages.

Superclass

PredefinedTypeElement

Semantics

12.13.2 IntegerUnit Class

The IntegerUnit is a meta-model element that represents integer data type common to various programming languages.

Superclass

PredefinedTypeElement

Semantics

12.13.3 CharUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.13.4 BooleanUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.13.5 FloatUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.13.6 FixedPointUnit Class

The FixedPointUnit is a meta-model element that represents fixed point data types common to various programming languages.

Superclass

PredefinedTypeElement

Semantics

12.13.7 DecimalUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.13.8 DateUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.13.9 TimeUnit Class

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

Superclass

PredefinedTypeElement

Semantics

12.14 DerivedTypes Class Diagram

The DerivedTypes class diagram defines meta-model elements that represent derived types, which are common to various programming languages. The classes and associations that make up the DerivedTypes diagram are shown in Figure 12.12.

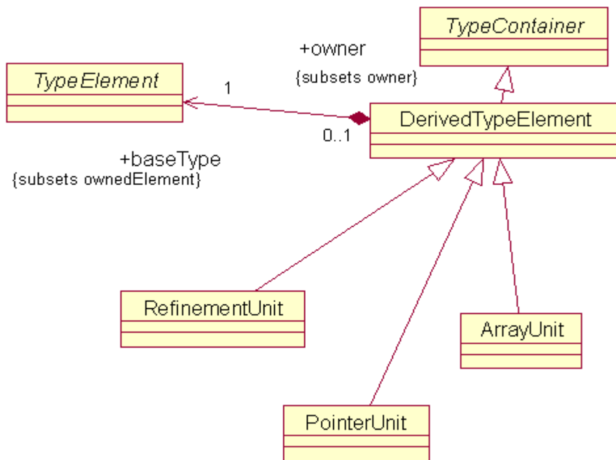

Figure 12.12 - DerivedTypes Class Diagram

12.14.1 DerivedTypeElement Class

The DerivedTypeElement is a meta-model element that represents user-defined types that are derived from a certain base type.

In the meta-model the RefinementType is a subclass of TypeContainer. It is associated with the corresponding base type. This class is subclassed by several more specific KDM classes.

Superclass

TypeContainer

Associations

baseType:TypeElement[1]	The TypeElement that is the base class of the derived type.

Constraints

Semantics

12.14.2 RefinementType Class

The RefinementType is a meta-model element that represents refinement types derived from a certain base type. In the meta-model the RefinementType is a subclass of DerivedTypeElement. It is associated with the corresponding base type.

Superclass

DerivedTypeElement

Constraints

Semantics

12.14.3 PointerType Class

The PointerType is a meta-model element that represents pointer datatypes. In the meta-model PointerType is a subclass of DerivedTypeElement.

Superclass

DerivedTypeElement

Constraints

Semantics

12.14.4 ArrayType Class

The ArrayType is a meta-model element that represents array datatypes. In the meta-model ArrayType is a subclass of DerivedTypeElement.

Superclass

DerivedTypeElement

Constraints

Semantics

12.15 EnumerationTypes Class Diagram

The EnumerationTypes class diagram defines meta-model elements that represent enumeration types, which are common to various programming languages. The classes and associations that make up the EnumerationTypes diagram are shown in Figure 12.13.

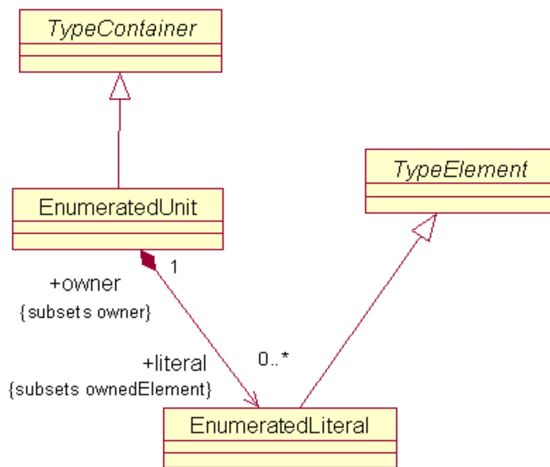

Figure 12.13 - EnumerationTypes Class Diagram

12.15.1 EnumeratedUnit Class

The EnumeratedUnit is a meta-model element that represents user-defined enumeration data types. EnumeratedUnit datatype defines the set of enumeration literals.

Superclass

TypeContainer

Associations

literal:EnumerationLiteral[1..*]	The set of enumeration literals defined for the target EnumerationType.

Semantics

12.15.2 EnumeratedLiteral Class

The EnumeratedLiteral is a meta-model element that represents enumeration literals.

Superclass

TypeElement

Constraints

Semantics

12.16 CompositeTypes Class Diagram

The CompositeTypes class diagram defines meta-model elements that represent common composite datatypes provided by various programming languages.

The classes and association that make up the StructuredTypes diagram are shown in Figure 12.14.

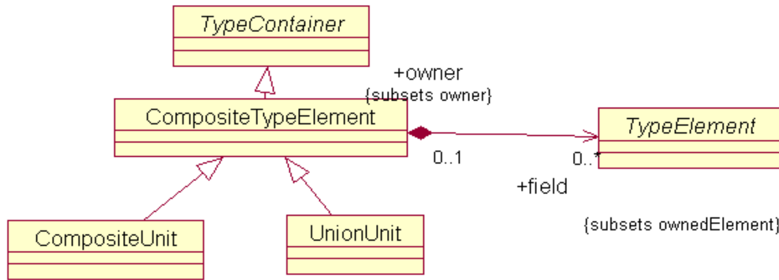

Figure 12.14 - CompositeTypes Class Diagram

12.16.1 CompositeTypeElement Class

The CompositeTypeElement is a meta-model element that represents user-defined composite datatypes, such as records, structures and unions. This element is further subclassed by more specific KDM classes.

Superclass

TypeContainer

Associations

field:TypeElement[0..*]	The set of contained datatypes.

Constraints

Semantics

12.16.2 UnionUnit Class

The UnionUnit is a meta-model element that represents user-defined union datatypes. A union datatype defines an ordered collection of variants. Each union variant is a named type element that is associated with a particular datatype.

Superclass

CompositeTypeElement

Constraints

Semantics

12.16.3 CompositeUnit Class

The CompositeUnit is a meta-model element that represents user-defined composite datatypes. A composite datatype defines an ordered collection of named fields where each field is a named element that has a specific data type.

Superclass

CompositeTypeElement

Constraints

Semantics

12.17 ClassTypes Class Diagram

The ClassTypes class diagram defines meta-model elements that represent common composite datatypes provided by various programming languages. The classes and association that make up the ClassTypes diagram are shown in Figure 12.15.

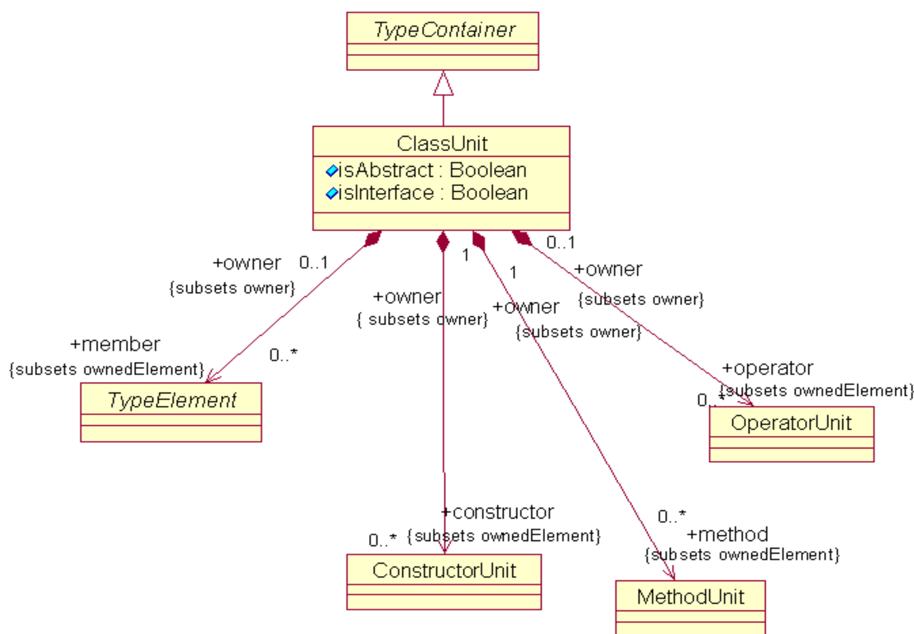

Figure 12.15 - ClassTypes Class Diagram

12.17.1 ClassUnit Class

The ClassUnit is a meta-model element that represents user-defined classes in object-oriented languages. A composite datatype defines an ordered collection of named fields where each field is a named element that has a specific data type.

Superclass

TypeContainer

Attributes

isAbstract:Boolean	The indicator of an abstract class.
isInterface:Boolean	The indicator of an interface class.

Associations

members:TypeElement[0..*]	The set of member datatypes
constructors:ConstructorUnit[1..*]	The set of constructors
methods:MethodUnit[0..*]	The set of methods
operators:OperatorUnit[0..*]	The set of operators

Constraints

Semantics

12.18 Signature Class Diagram

The Signature class diagram defines meta-model elements that represent the signature concept common to various programming languages. The classes and associations that make up Signature diagram are shown in Figure 12.16.

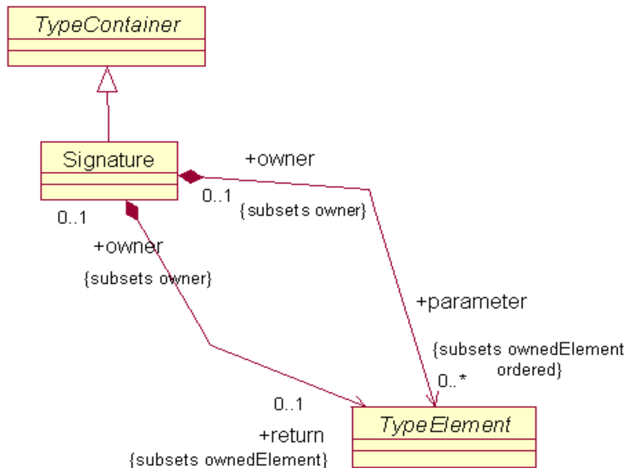

Figure 12.16 - Signature Class Diagram

12.18.1 Signature Class

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

Superclass

TypeContainer

Associations

parameter:TypeElement[0..*]	The set of parameter types that are associated with the current Signature.
return:TypeElement[0..1]	The return type for the current Signature.

Constraints

Semantics

12.19 Interface Class Diagram

The Interface class diagram defines meta-model elements that represent the concept of an interface common to various programming languages. The classes and association that make up the Interface diagram are shown in Figure 12.17.

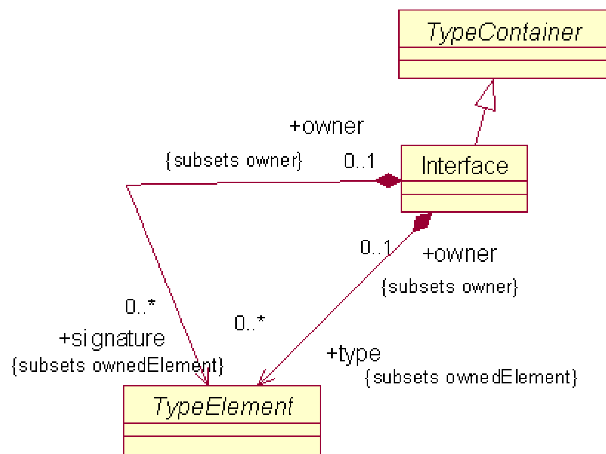

Figure 12.17 - Interface Class Diagram

12.19.1 Interface Class

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

In the meta-model Interface is a subclass of TypeContainer. Interface is a set of TypeElements (representing data types and Signatures).

Superclass

TypeContainer

Associations

type:TypeElement[0..*]	The set of TypeElements which are associated with the target Interface.
signature:TypeElement[0..*]	The set of signatures which are associated with the target Interface.

Constraints

Semantics

12.20 InterfaceRelations Class Diagram

The Interface class diagram defines KDM relationships that are related to the concept of an interface. The classes and association that make up the Interface diagram are shown in Figure 12.18.

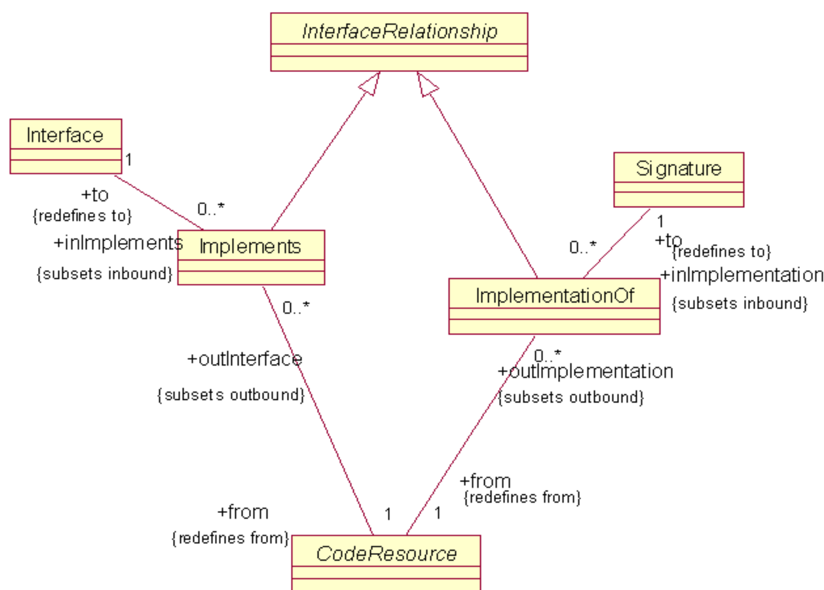

Figure 12.18 - InterfaceRelations Class Diagram

12.20.1 Implements Class

The Implements is a meta-model element that represents “implementation” relation between a CodeResource (for example, a ClassUnit) and an Interface. In the meta-model Implements is a subclass of InterfaceRelationship.

Superclass

InterfaceRelationship

Associations

from:CodeResource[1]	The CodeResource which implements Interface.
to:Interface[1]	The Interface which is being implemented by CodeResource.

Constraints

Semantics

12.20.2 ImplementationOf Class

The ImplementationOf is a meta-model element that represents "implementation" relation between a CodeResource (for example, a MethodUnit) and a Signature. In the meta-model ImplementationOf is a subclass of InterfaceRelationship.

Superclass

InterfaceRelationship

Associations

from:CodeResource[1]	The CodeResource which implements Interface.
to:Signature[1]	The Signature which is being implemented by CodeResource.

Constraints

Semantics

12.20.3 CodeResource (additional properties)

Associations

outInterface:Implements[0..*]

OutImplementation:ImplementationOf[0..*]

Constraints

Semantics

12.20.4 Interface (additional properties)

Associations

inImplements:Implements[0..*]

Constraints

12.20.5 Signature (additional properties)

Associations

inImplementation:ImplementationOf[0..*]	

Constraints

12.21 TypeRelations Class Diagram

The TypeRelations class diagram defines meta-model elements that represent semantic associations between datatypes and data elements. The classes and associations that make up the TypeRelations diagram are shown in Figure 12.19.

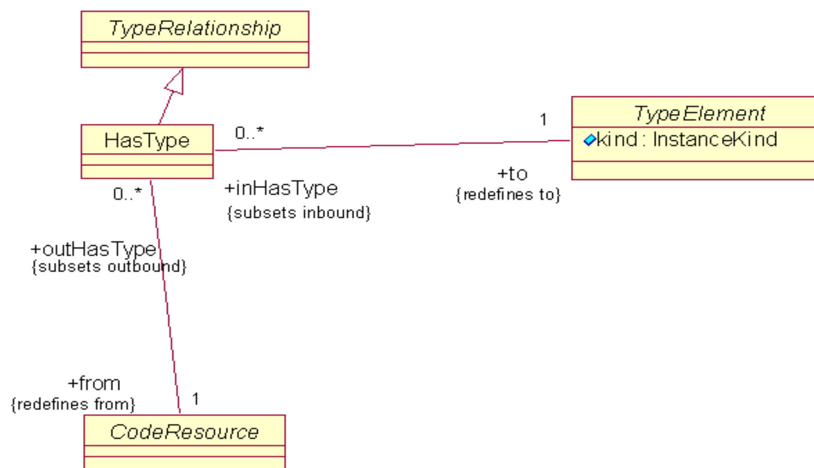

Figure 12.19 - TypeRelations Class Diagram

12.21.1 HasType Class

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

Superclass

TypeRelationship

Associations

from:CodeElement[1]	The source data element (represented by a DataInterface).
to:TypeElement[1]	The target type element (represented by a TypeElement).

Constraints

Semantics

12.21.2 CodeResource (additional properties)

Associations

outHasType:HasType[0..*]

Constraints

Semantics

12.21.3 TypeElement (additional properties)

Associations

inHasType:HasType[0..*]

Constraints

Semantics

12.22 ClassRelations Class Diagram

The ClassRelations class diagram defines meta-model elements that represent semantic associations between datatypes and data elements. The classes and associations that make up the ClassRelations diagram are shown in Figure 12.20.

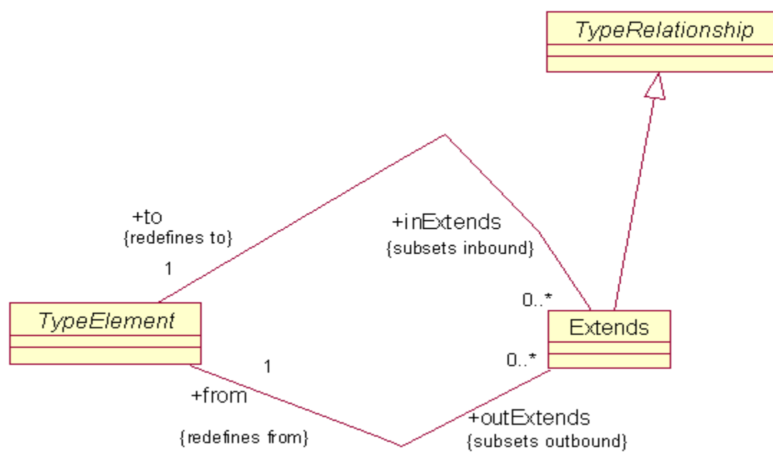

Figure 12.20 - ClassRelations Class Diagram

12.22.1 Extends Class

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

In the meta-model Extends is a subclass of TypeRelationship.

Superclass

TypeRelationship

Associations

from:TypeElement[1]	The subclass Class (represented by a TypeElement)
to:TypeElement[1]	The parent Class (represents by a TypeElement)

Constraints

Semantics

12.22.2 TypeElement (additional properties)

Associations

outExtends:Extends[0..*]	The set of outbound Extends relations
inExtends:Extends[0..*]	The set of inbound Extends relations

Constraints

Semantics

12.23 Comment Class Diagram

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

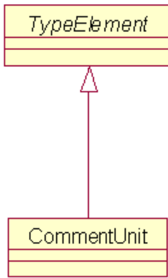

Figure 12.21 - Comment Class Diagram

12.23.1 CommentUnit Class

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

In the meta-model Extends is a subclass of TypeRelationship.

Superclass

TypeElement

Constraints

Semantics

12.24 Visibility Class Diagram

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

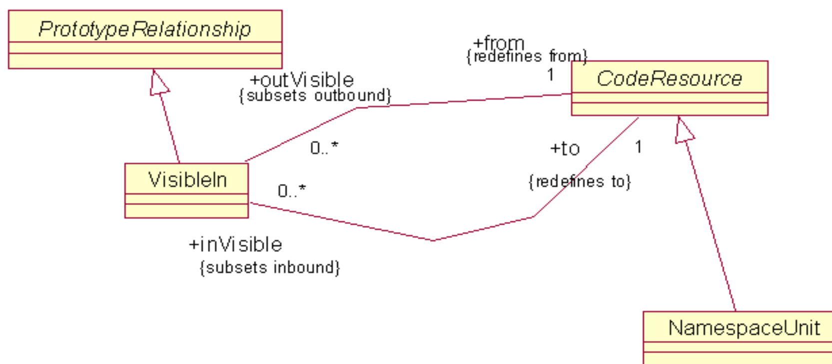

Figure 12.22 - Visibility Class Diagram

12.24.1 VisibleIn Class

The VisibleIn is a specific meta-model element that represents semantic relation between two code resources, where one provides the restricted visibility context for another resource. In the meta-model VisibleIn is a subclass of PrototypeRelationship.

Superclass

PrototypeRelationship

Associations

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

Constraints

Semantics

12.24.2 Namespace Class

The Namespace is a specific meta-model element that represents can be the target of the VisibleIn KDM relation.

Superclass

CodeResource

Constraints

Semantics

12.24.3 CodeResource (additional properties)

Associations

outVisible:VisibleIn[0..*]	The set of outbound VisibleIn relations
inVisible:VisibleIn[0..*]	The set of inbound VisibleIn relations

Constraints

Semantics

13 Action Package

13.1 Overview

The Action package contains model classes and associations capturing system artifacts to model the behavior of programming languages such as statements, features, business rules, and their relationships. It includes specific classes to model behavior related to calls, data accesses, prototypes, and control-flow.

13.2 Organization of the Action Package

The Action package is a collection of classes and associations that are described together because they provide meta-model constructs for defining behavior of programming languages artifacts and their relationships to the Code package.

The Action package consists of the following class diagrams:

- ActionModel
- ActionFlow
- ActionRelations
- CallableRelations
- DataRelations
- ImportRelations
- MacroRelations
- TypeRelations
- PrototypeRelations

The Action package depends on the following packages:

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Source

org.omg::ADM::KDM::Core

13.3 ActionRelations Class Diagram

The ActionRelations class diagram abstract classes for representing several families of KDM relations. The class diagram shown in Figure 13.1 captures these classes and their relations.

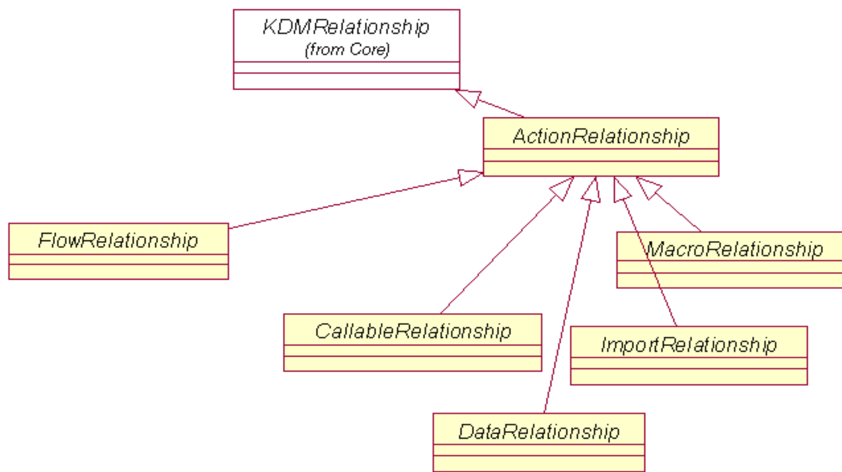

Figure 13.1 - ActionRelations Class Diagram

13.3.1 ActionRelationship Class (abstract)

The ActionRelationship is the parent class representing various relations involving ActionElements.

Superclass

KDMRelationship

Constraints

Semantics

13.3.2 FlowRelationship Class (abstract)

The FlowRelationship class represents the family of control flow relations between ActionElements.

Superclass

ActionRelationship

Constraints

Semantics

13.3.3 MacroRelationship Class (abstract)

The MacroRelationship class represents the family of relations corresponding to the usage of MacroUnits.

Superclass

ActionRelationship

Constraints

Semantics

13.3.4 CallableRelationship Class (abstract)

The CallableRelationship class represents the family of relations between ActionElements and CallableElements.

Superclass

ActionRelationship

Constraints

Semantics

13.3.5 DataRelationship Class (abstract)

The DataRelationship class represents the family of relations between ActionElements and TypeElements.

Superclass

ActionRelationship

Constraints

Semantics

13.3.6 ImportRelationship Class (abstract)

The ImportRelationship class represents the family of relations corresponding to import of declarations between modules.

Superclass

ActionRelationship

Constraints

Semantics

13.4 ActionModel Class Diagram

The ActionModel class diagram provides basic meta-model constructs to define basic unit of behavior and allows grouping.

ActionElements are part of CodeModels. ActionElements can be contained in some CodeContainers as well as in CallableUnits.

The class diagram shown in Figure 13.2 captures these classes and their relations.

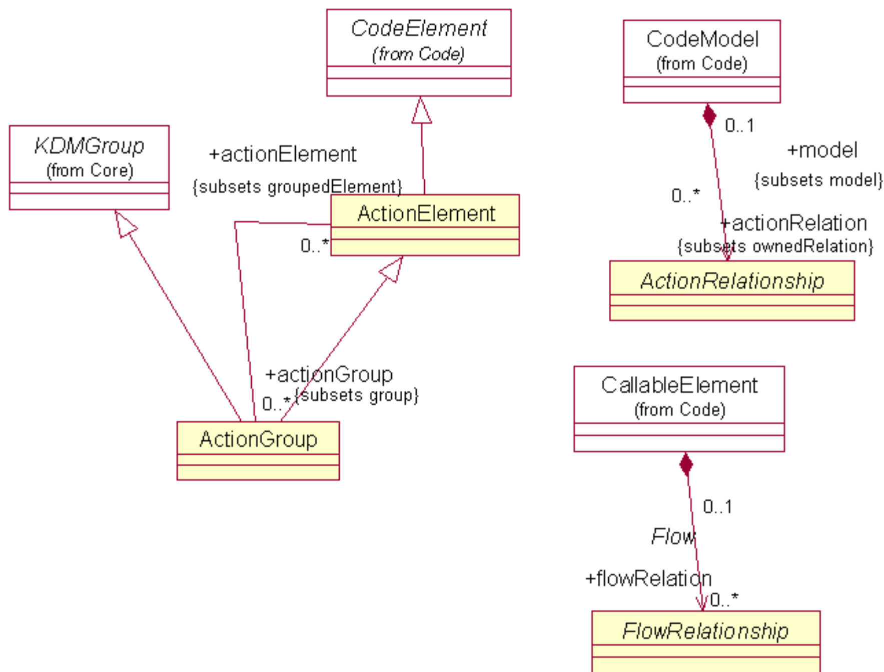

Figure 13.2 - ActionModel Class Diagram

13.4.1 ActionElement Class

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

Superclass

CodeElement

Associations

actionGroup:ActionGroup[0..*]

Constraints

Semantics

13.4.2 ActionGroup Class

The ActionGroup is a grouping of ActionElement.

Superclass

KDMGroup

ActionElement

Associations

groupedElement:ActionElement[0..*]

Constraints

Semantics

13.4.3 CallableElement (additional properties)

Associations

flowRelation:FlowRelationship[0..*]

Constraints

Semantics

13.4.4 CodeModel (additional properties)

Associations

actionRelation:ActionRelationship[0..*]

Constraints

Semantics

13.5 ActionFlow Class Diagram

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

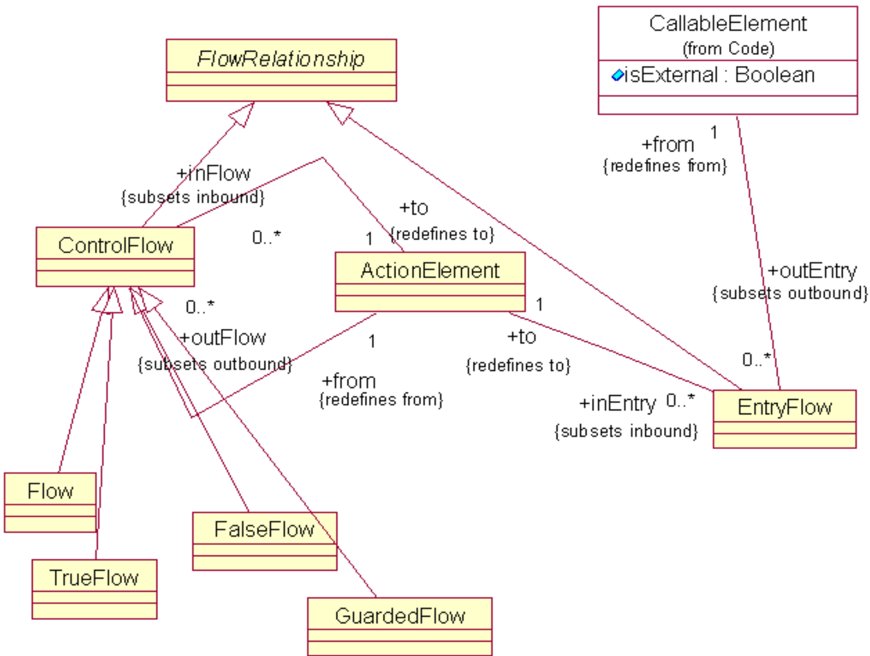

Figure 13.3 - ActionFlow Class Diagram

13.5.1 ControlFlow Class

The ControlFlow is an association class representing the control flow between ActionElements.

Superclass

ActionRelationship

Associations

from:ActionElement[1]

to:ActionElement[1]

Constraints

Semantics

13.5.2 EntryFlow Class

The ControlFlow is an association class representing the control flow between ActionElements.

Superclass

ActionRelationship

Associations

from:CallableElement[1]

to:ActionElement[1]

Constraints

Semantics

13.5.3 ActionElement Class (additional properties)

Associations

inFlow:ControlFlow[0..*]

outFlow:ControlFlow[0..*]

inEntry:EntryFlow[0..*]

Constraints

Semantics

13.5.4 CallableElement Class (additional properties)

Associations

outEntry:EntryFlow[0..*]

Constraints

Semantics

13.5.5 Flow Class (abstract)

The ImportRelationship class represents the family of relations corresponding to import of declarations between modules.

Superclass

ActionRelationship

Constraints

Semantics

13.5.6 TrueFlow Class (abstract)

The ImportRelationship class represents the family of relations corresponding to import of declarations between modules.

Superclass

ActionRelationship

Constraints

Semantics

13.5.7 FalseFlow Class (abstract)

The ImportRelationship class represents the family of relations corresponding to import of declarations between modules.

Superclass

ActionRelationship

Constraints

Semantics

13.5.8 GuardedFlow Class (abstract)

The ImportRelationship class represents the family of relations corresponding to import of declarations between modules.

Superclass

ActionRelationship

Constraints

Semantics

13.6 CallableRelations Class Diagram

The CallableRelations class diagram collects together classes and associations of the Action package. They provide basic meta-model constructs to define call type behavior and link ActionElement to CodeElement.

The CallableRelations diagram describes the following types:

- CodeRelationship – an association class subtyping KDMRelationship.
- Calls – an association class representing a call type relationship between an ActionElement and a CallableInterface.
- ActionElement – a class representing a basic unit to describe behavior.
- UsesCode – a class representing an association class to link ActionElement to CodeElement.

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

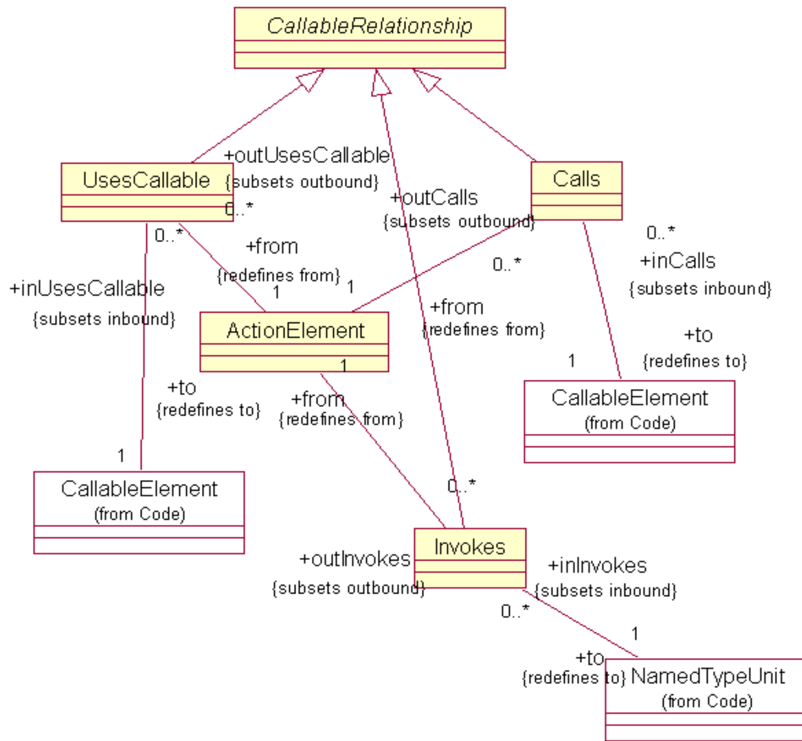

Figure 13.4 - CallableRelations Class Diagram

13.6.1 Calls Class

The Calls class is a subtype of CodeRelationship and provides linkages between ActionElement and CallableInterface for the situations when an ActionElement performs a call to the CallableElement.

Superclass

CallableRelationship

Associations

from:ActionElement[1]

to:CallableElement[1]

Constraints

Semantics

13.6.2 UsesCallable Class

The UsesCallable is an association class to provide linkages between ActionElement and CodeElement for any references to a CallableElement, other than performing a call.

Superclass

CallableRelationship

Associations

from:ActionElement[1]

to:CodeElement[1]

Constraints

Semantics

13.6.3 Invokes Class

The Invokes represents a relationship between an ActionElement and a NamedType when an ActionElement accesses an instance of a class to invoke a method or access an attribute of a class.

Superclass

CallableRelationship

Associations

from:ActionElement[1]

to:NamedTypeElement[1]

Constraints

Semantics

13.6.4 CallableElement (additional properties)

Associations

inCalls:Calls[0..*]

inUsesCallable:UsesCallable[0..*]

Constraints

Semantics

13.6.5 ActionElement Class (additional properties)

Associations

outCalls:Calls[0..*]

outUsesCallable:UsesCallable[0..*]

outInvokes:Invokes[0..*]

Constraints

Semantics

13.6.6 NamedTypeElement (additional properties)

Associations

inInvokes:Invokes[0..*]

Constraints

Semantics

13.7 DataRelations Class Diagram

The DataRelations class diagram collects together classes and associations of the Action package. They provide basic meta-model constructs to define data specific CRUD like relationships between ActionElement and DataInterface. The class diagram shown in Figure 13.5 captures these classes and their relations.

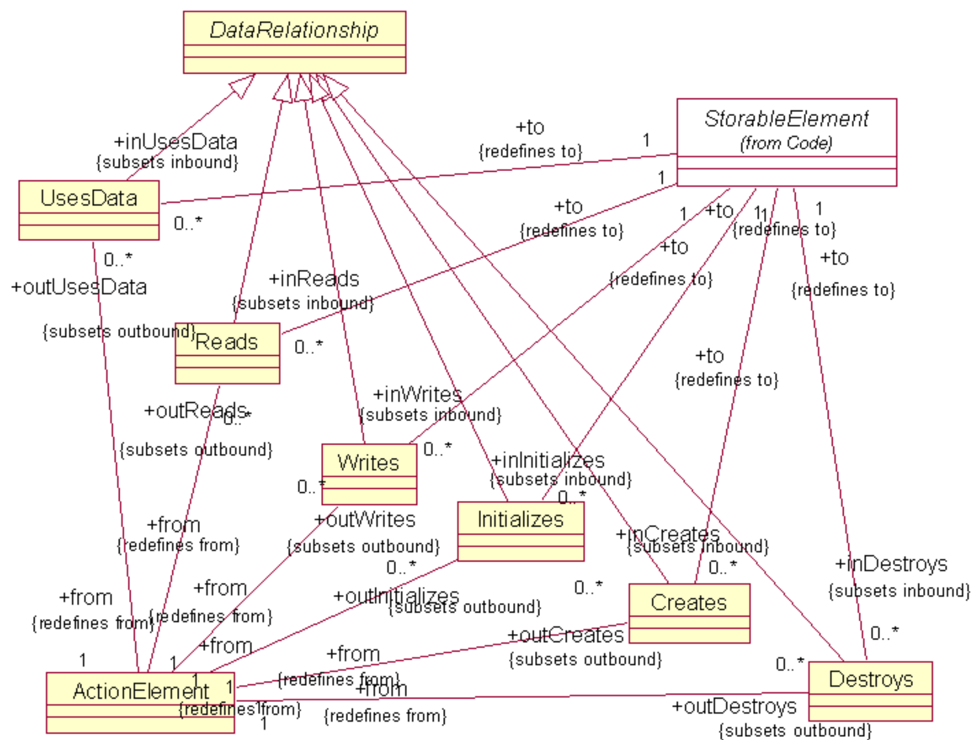

Figure 13.5 - DataRelations Class Diagram

13.7.1 Reads Class

The Reads association provides linkages between ActionElement and StorableElement modeling a read access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.2 Writes Class

The Writes association provides linkages between ActionElement and StorableElement modeling a write access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.3 UsesData Class

The UsesData association provides linkages between ActionElement and StorableElement modeling a use access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.4 Creates Class

The Creates association provides linkages between ActionElement and StorableElement modeling a create access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.5 Destroys Class

The Destroys association provides linkages between ActionElement and StorableElement modeling a destroy access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.6 Initializes Class

The Initializes association provides linkages between ActionElement and StorableElement modeling an initialization access.

Superclass

DataRelationship

Associations

from:ActionElement[1]

to:StorableElement[1]

Constraints

Semantics

13.7.7 StorableElement (additional properties)

Associations

inReads:Reads[0..*]

inWrites:Writes[0..*]

inUsesData:UsesData[0..*]

inCreates:Creates[0..*]

inDestroys:Destroys[0..*]

inInitializes:Initializes[0..*]

Constraints

Semantics

13.7.8 ActionElement Class (additional properties)

Associations

outReads:Reads[0..*]

outWrites:Writes[0..*]

outUsesData:UsesData[0..*]

outCreates:Creates[0..*]

outDestroys:Destroys[0..*]

outInitializes:Initializes[0..*]

Constraints

Semantics

13.8 PrototypeRelations Class Diagram

The PrototypeRelations class diagram defines basic meta-model constructs to model prototype relationships between ActionElement and CodeElement. The class diagram shown in Figure 13.6 captures these classes and their relations.

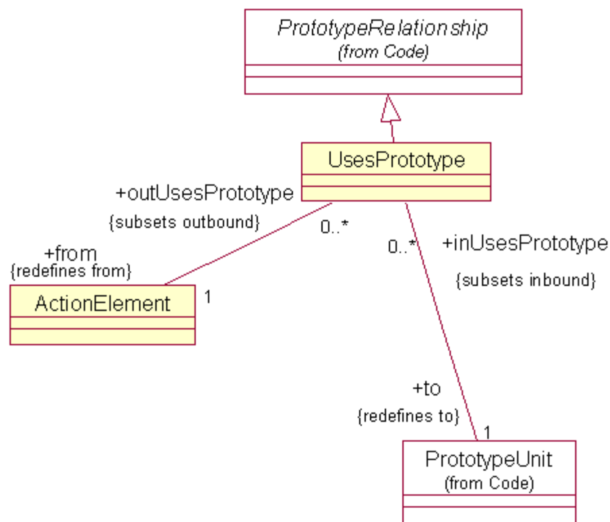

Figure 13.6 - PrototypeRelations Class Diagram

13.8.1 UsesPrototype Class

The UsesPrototype is an association class to provide linkages between ActionElement and PrototypeUnit.

Superclass

PrototypeRelationship

Associations

from:ActionElement[1]

to:PrototypeUnit[1]

Constraints

Semantics

13.8.2 ActionElement Class (additional properties)

Associations

outUsesPrototype:UsesPrototype[0..*]

Constraints

Semantics

13.8.3 PrototypeUnit Class (additional properties)

Associations

inUsesPrototype:UsesPrototype[0..*]

Constraints

Semantics

13.9 ImportRelations Class Diagram

The ImportRelations class diagram provides basic meta-model constructs to define relationships between ImportDirective and CodeResource

The class diagram shown in Figure 13.7 captures these classes and their relations.

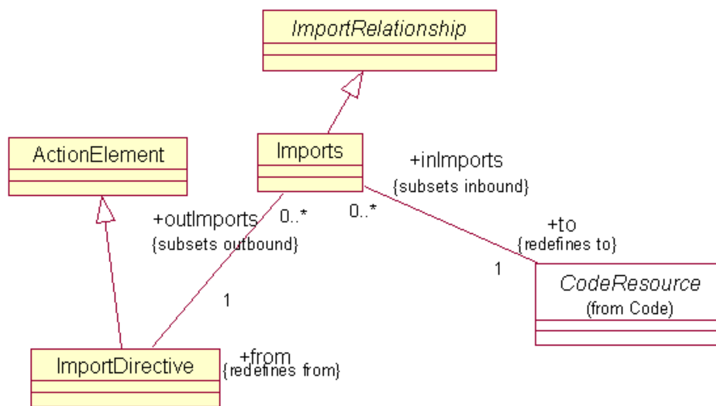

Figure 13.7 - ImportRelations Class Diagram

13.9.1 ImportDirective Class

The ImportDirective is a subtype of ActionElement representing the import concept used in object-oriented languages.

Superclass

ActionElement

Associations

outImports:Imports[0..*]

Constraints

Semantics

13.9.2 Imports Class

The Imports association provides linkages between ImportDirective and CodeResource.

Superclass

ImportRelationship

Associations

from:ImportDirective[1]

to:CodeResource[1]

Constraints

Semantics

13.9.3 CodeResource Class (additional properties)

Associations

inImports:Imports[0..*]

Constraints

Semantics

13.10 TypeRelations Class Diagram

The TypeRelations class diagram collects together classes and associations of the Action package. They provide basic meta-model constructs to define relationships between ActionElement and TypeElement. The class diagram shown in Figure 13.8 captures these classes and their relations.

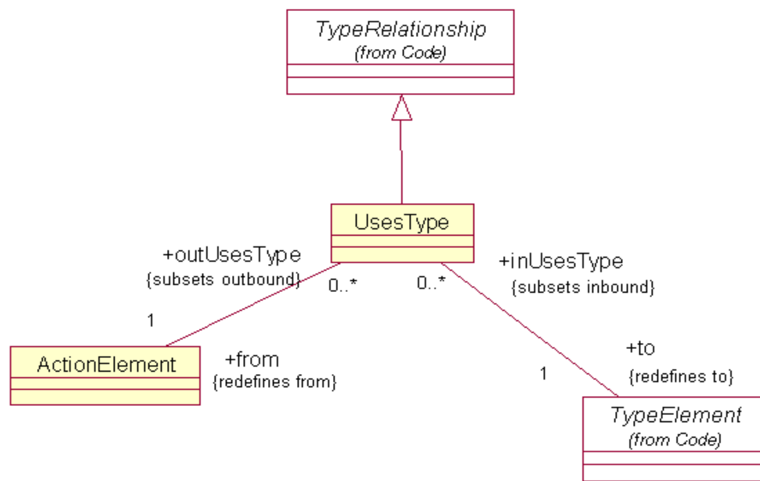

Figure 13.8 - TypeRelations Class Diagram

13.10.1 UsesType Class

The UsesType association provides linkages between ActionElement and TypeElement.

Superclass

TypeRelationship

Associations

from:ActionElement[1]

to:TypeElement[1]

Constraints

Semantics

13.10.2 ActionElement Class (additional properties)

Associations

outUsesType:UsesType[0..*]

Constraints

Semantics

13.10.3 TypeElement Class (additional properties)

Associations

inUsesType:UsesType[0..*]

Constraints

Semantics

13.11 MacroRelations Class Diagram

The MacroRelations class diagram provides basic meta-model constructs to define relationships between ActionElement and MacroUnit. The class diagram shown in Figure 13.9 captures these classes and their relations.

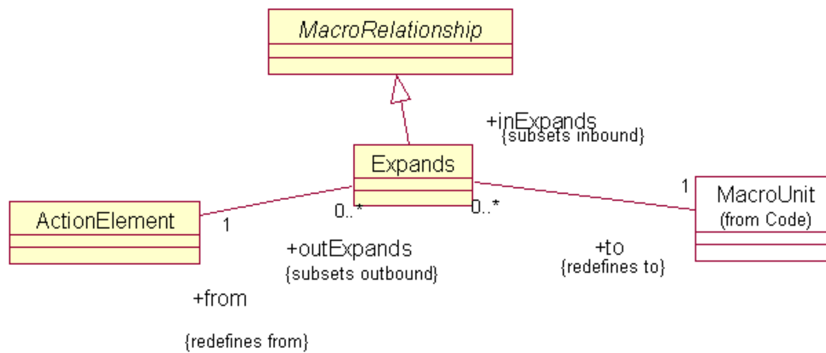

Figure 13.9 - MacroRelations Class Diagram

13.11.1 Expands Class

The Expands association provides linkages between ActionElement and MacroUnit.

Superclass

MacroRelationship

Associations

from:ActionElement[1]

to:MacroUnit[1]

Constraints

The endpoint can be only MacroUnit

Semantics

13.11.2 MacroUnit Class (additional properties)

Associations

inExpands:Expands[0..*]

Constraints

Semantics

14 Build Package

14.1 Overview

The Build package represents the artifacts that model the engineering view of a particular existing system. The Build package also includes the entities to model objects that are generated by the build process.

14.2 Organization of the Build Package

The Build package is a collection of classes and associations that are described together because they provide a meta-model for defining the constructs required to model the building of a software system.

The Build package consists of the following class diagrams:

- BuildInheritances
- BuildModel
- BuildResources
- BuildRelations

The Build package depends on the following packages:

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Core

14.3 BuildInheritances Class Diagram

The BuildInheritances Class Diagram shown in Figure 14.1 depicts how various build classes extend other KDM classes. Each of the classes shown is this diagram inherits properties from classes found in the KDM Core package.

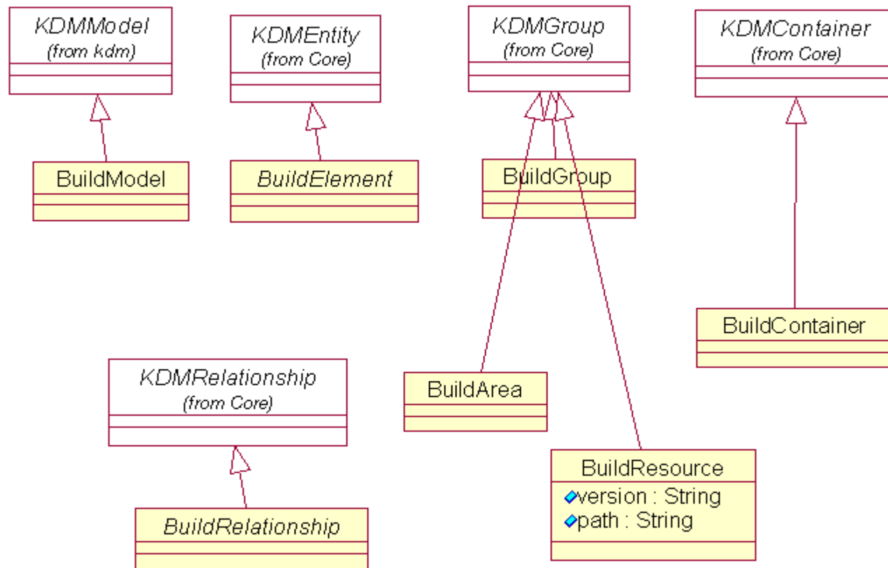

Figure 14.1 - BuildInheritances Class Diagram

14.4 BuildModel Class Diagram

The BuildModel class diagram provides basic meta-model constructs to model the engineering view of a particular existing software system within the KDM framework. The class diagram shown in Figure 14.2 captures these classes and their relations.

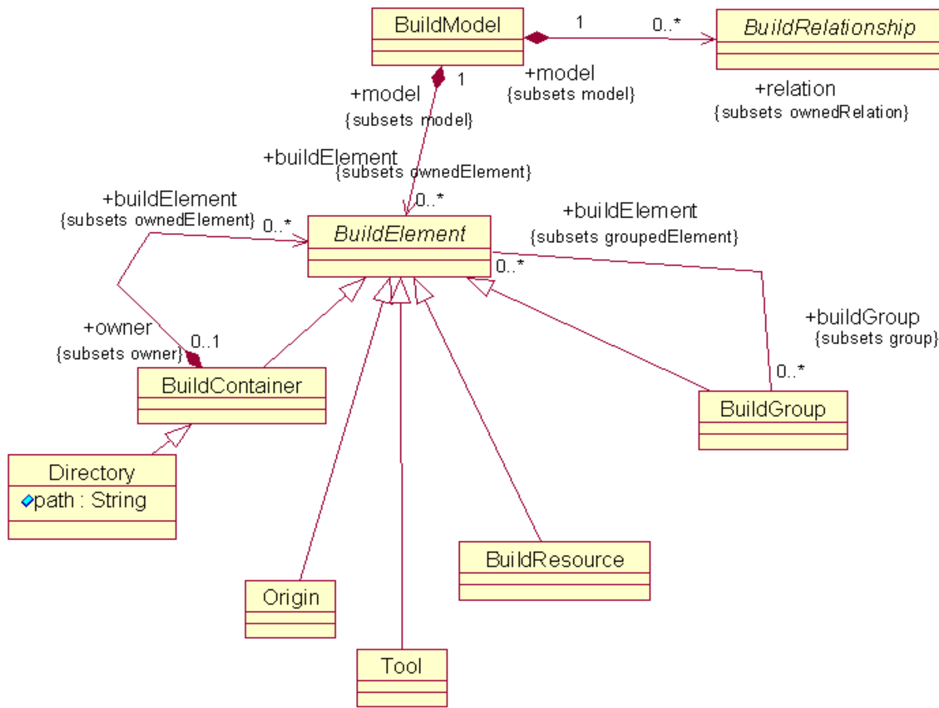

Figure 14.2 - BuildModel Class Diagram

14.4.1 BuildModel Class

The BuildModel encapsulates meta-model constructs needed to model the building of a particular software system.

Superclass

KDMModel

Associations

buildElement:BuildElement[0..*]	The set of BuildElements
Relation:BuildRelation[0..*]	The set of build relations

Constraints

Semantics

14.4.2 BuildElement Class

The BuildElement is the abstract base class from which all other build model elements are extended.

Superclass

KDMEntity

Associations

model:BuildModel[1]	
owner:BuildContainer[0..1]	
buildGroup:BuildGroup[0..*]	

Constraints

Semantics

14.4.3 BuildGroup Class

The BuildGroup is a container class, which is used to aggregate groups of build elements.

Superclass

BuildElement

KDMContainer

Associations

elements:BuildElement[0..*]	BuildElement is the class from which all other build model elements are extended.

Constraints

Semantics

14.4.4 BuildResource Class

The BuildResource class is the parent class, which represents build entities. This class is then extended to concrete entities representing artifacts such as source files. Build entities are groups for code elements.

Superclass

BuildElement

KDMGroup

Attributes

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

Semantics

14.4.5 Directory Class

The Directory class represents directories as they are used to group together related files.

Superclass

BuildContainer

Attributes

path:String	Location of the directory.

Semantics

14.4.6 Origin Class

The Origin class models producers of the 3rd party software components as they contribute to the build process.

Superclass

BuildElement

Semantics

14.4.7 Tool Class

The Tool class represents software tools as they are used in the build process.

Superclass

BuildElement

Semantics

14.5 BuildResources Class Diagram

The BuildResources class diagram provides basic meta-model constructs to model various common build resource and their relations to the code model. The class diagram shown in Figure 14.3 captures these classes and their relations.

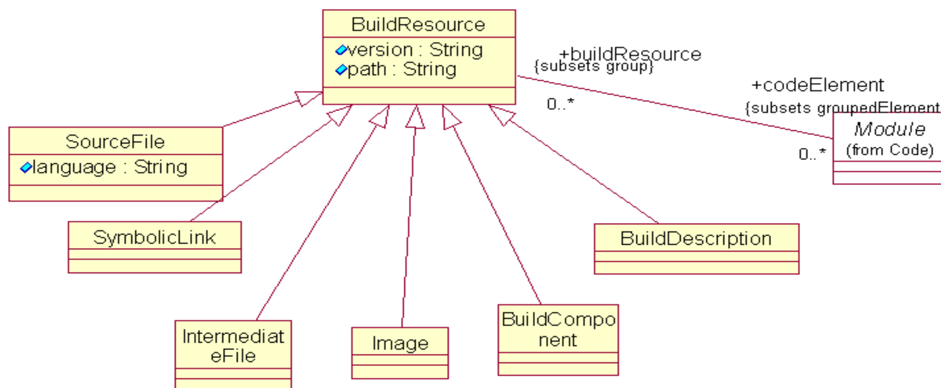

Figure 14.3 - BuildResources Class Diagram

14.5.1 BuildResource Class (additional properties)

Associations

codeElement:Module[0..*]	BuildResource references code elements

Semantics

14.5.2 SourceFile Class

The SourceFile class represents source files. This class can be used to provide links between code elements and their physical implementations using the SourceRegion mechanism from the Source package.

Superclass

BuildResource

Attributes

language:String	Indicates the language of the source file.

Semantics

14.5.3 IntermediateFile Class

The IntermediateFile represents intermediate binary files, for example relocatable object files.

Superclass

BuildResource

Semantics

14.5.4 BuildComponent Class

The BuildComponent class represents binary files that correspond to deployable components, for example executable files.

Superclass

BuildResource

Semantics

14.5.5 BuildDescription Class

The BuildDescription class is used to model objects such as make files or ant scripts, which describe the build process itself.

Superclass

BuildResource

Semantics

14.5.6 SymbolicLink Class

The SymbolicLink is used to represent symbolic links.

Superclass

BuildResource

Semantics

14.5.7 Image Class

The Image is used to represent image files.

Superclass

BuildResource

Semantics

14.6 BuildRelations Class Diagram

The BuildRelations class diagram defines the various build related relationships.

The class diagram shown in Figure 14.4 captures these classes and their relations.

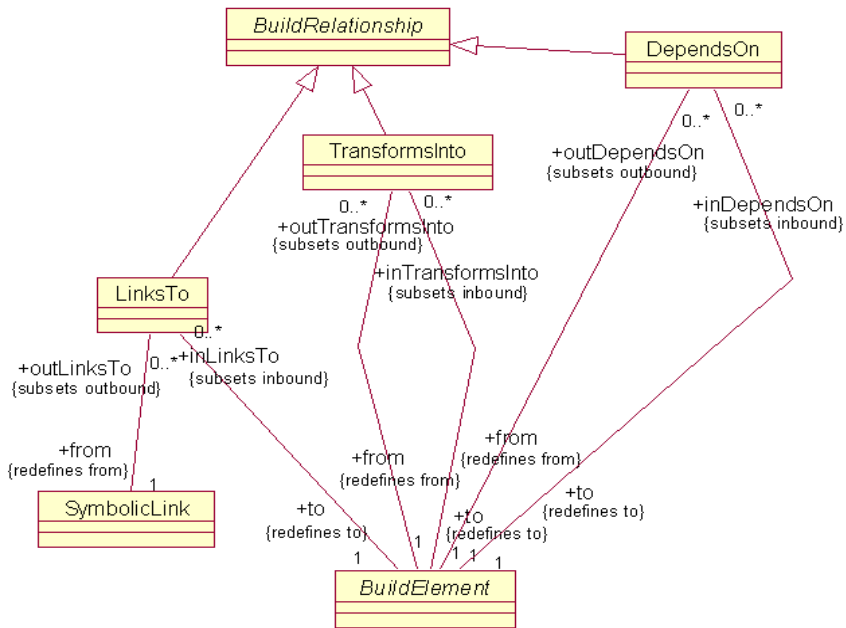

Figure 14.4 - BuildRelations Class Diagram

14.6.1 BuildRelationship Class (abstract)

The BuildRelationship class is the abstract base class from which all other build relationship classes are extended.

Superclass

KDMRelationship

Semantics

14.6.2 LinksTo Class

The LinksTo class models the relationship between two linked build resources.

Superclass

BuildRelationship

Associations

from:SymbolicLink[1]

to:BuildElement[1]

Constraints

Semantics

14.6.3 DependsOn Class

The DependsOn class models build dependencies between build elements. These dependencies can be to third party components or other components within the same system.

Superclass

BuildRelationship

Associations

from:BuildElement[1]

to:BuildElement[1]

Constraints

Semantics

14.6.4 GeneratedBy Class

The GeneratedBy class models the relationship between build elements and the artifacts they produce as a result of the build process.

Superclass

BuildRelationship

Associations

from:BuildElement[1]

to:BuildElement[1]

Constraints

Semantics

14.6.5 BuildElement (additional properties)

Associations

inLinksTo:LinksTo[0..*]

inGeneratedBy:GeneratedBy[0..*]

outGeneratedBy:GeneratedBy[0..*]

inDependsOn:DependsOn[0..*]

outDependsOn:DependsOn[0..*]

Constraints

Semantics

14.6.6 SymbolicLink (additional properties)

Associations

outLinksTo:LinksTo[0..*]

Constraints

Semantics

15 Data Package

15.1 Overview

The KDM Data Package describes the meta-model elements that represent persistent data within a system.

15.2 Organization of the Data Package

The Data package describes KDM classes and associations that represent persistent data aspects of enterprise applications within the common KDM framework.

The Data package consists of the following class diagrams:

- Data Inheritance
- Data Model
- KeyIndex
- KeyRelations
- RelationalData
- ColumnSet
- RecordData
- XMLData
- XMLElements
- ProgramElements

The Data Package depends on the following packages:

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Core

15.3 Data Inheritance

The Data Inheritance Diagram shown in Figure 15.1 depicts how various data classes derive from the Core KDM classes. Each of the Data Package classes within this diagram inherits certain properties from KDM classes defined within the KDM Core Package.

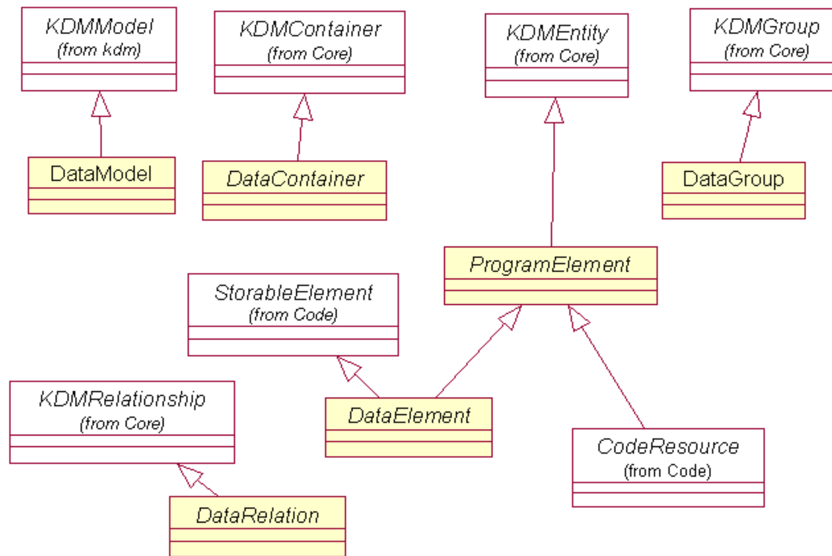

Figure 15.1 - DataInheritance Diagram

The Data Inheritance Diagram in Figure 15.1 depicts the following properties of inheritance from the KDM Model.

- DataModel – A class representing a type of KDM Model that owns logical representation of data within a system.
- DataContainer – A class representing a type of KDM Container. Within the Data Model, a Data Container contains data found within a system.
- DataElement – A class representing a type of KDM Entity that was derived from the analysis of data representations within a system. A Data Element is the centerpiece of the Data Package and discussed further later in this chapter. DataElement inherits from a ProgramElement class (from Code package) as well as from StorableElement class (from Action package).
- DataGroup – A class representing a type of KDM Group where data or entities are collected and grouped together. A Data Group is a special kind of element that contains other elements. Data Groups are also discussed later in this chapter.
- DataRelation – A class representing a type of KDM Relationship where a key is used to access data in ways other than sequentially accessing data. Relations are established via Keys and are discussed later in this chapter.

15.4 Data Model Class Diagram

The Data Model Diagram in Figure 15.2 depicts the central concept of a Data Element, types of Data Elements.

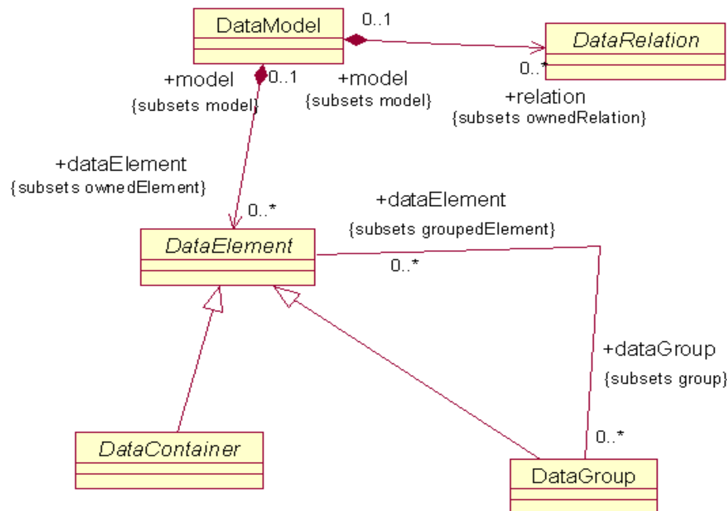

Figure 15.2 - Data Model

15.4.1 DataModel Class

The DataModel Class represents a logical Data Model of an existing system. A Data Model is composed of Data Elements, as shown in Figure 15.2.

Superclass

KDMModel

Associations

dataElement :DataElement[0..*]	
relation:DataRelation[0..*]	Data relations that are owned by this Data Model

Constraints

Semantics

15.4.2 DataElement Class

The DataElement Class represents the discreet instance of a given data element within a system. For example, a *Customer_Number* is one type of data element that might be found within a system. Data Element interfaces with the Code Package through the class entitled Program Element. DataElement interfaces with the Code Package through the class called StorableElement.

Superclass

KDMEntity

Code::StorableElement

ProgramElement

Associations

model: DataModel[1]	The model that owns the current element
dataGroup:DataGroup[0..*]	The set of groups to which the current element belongs

Constraints

Semantics

15.4.3 DataGroup Class

A Data Group represents a collection of data elements. A Data Group results when an analyzer finds a Data Element that contains other Data Elements. Artifacts from which a group would be derived include tables, records, segments and related structures that represent a collection of data elements. A Data Group can be a part of another Data Group. A DataElement can be associated with multiple DataGroups.

Superclass

KDMGroup

DataElement

Associations

dataElement :DataElement[0..*]	

Constraints

Semantics

15.4.4 DataContainer Class

A DataContainer Class is a type of DataElement. A Data Container owns Data Elements, as shown in Figure 15.2

Superclass

KDMContainer

DataElement

Constraints

Semantics

15.5 KeyIndex Class Diagram

The KeyIndex class diagram collects together classes and associations of the Data package. They provide basic meta-model constructs to define the various data related relationships.

The KeyIndex diagram describes the following types:

- BuildRelationship – an abstract class representing a generic build relationship.
- LinksTo – a class representing a build resource, which is linked to another, build resource as in a symbolic link in UNIX.
- DependsOn – a class representing a build dependency such as a dependency on a particular third party library.
- GeneratedBy – a class representing an artifact, which is produced as a result of the build process.

The class diagram shown in Figure 15.3 captures these classes and their relations.

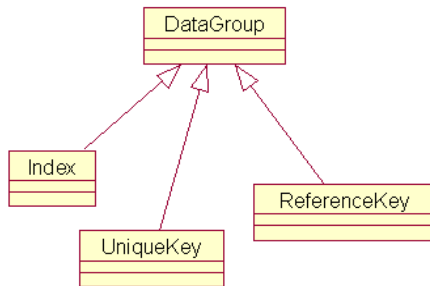

Figure 15.3 - KeyIndex Class Diagram

15.5.1 Index Class

An Index is a way to access data within an array or table. An Index is a type of Data Group. An Index is also a type of Data Element.

Superclass

DataGroup

Semantics

15.5.2 UniqueKey Class

A UniqueKey is a way to define one or more Data Elements as a unique key into an array or table. A UniqueKey is a type of Data Group.

Superclass

DataGroup

Constraints

Semantics

15.5.3 ReferenceKey Class

A ReferenceKey is a way to access unique keys within an array or table. A ReferenceKey is a type of Data Group.

Superclass

DataGroup

Constraints

Semantics

15.6 RelationalData Class Diagram

The RelationalData class diagram provides basic meta-model constructs to relational databases within the KDM framework. The class diagram shown in Figure 15.4 captures these classes and their relations.

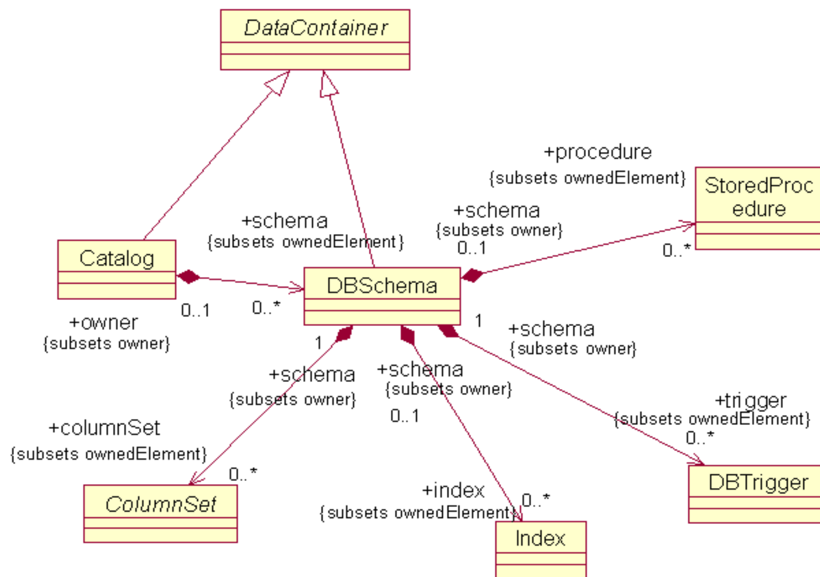

Figure 15.4 - RelationalData Class Diagram

15.6.1 Catalog Class

The Catalog class is the top level container that represents a relational database.

Superclass

DataContainer

Associations

schema :DBSchema[0..*]	Schemas of relational database that are owned by this catalog.

Semantics

15.6.2 DBSchema Class

The DBSchema class is a relational database schema.

Superclass

DataContainer

Associations

columnSet :ColumnSet[0..*]	Tables and Views owned by this schema
index:Index[0..*]	Indices owned by this schema
trigger:DBTrigger[0..*]	
procedure:StoredProcedure[0..*]	

Constraints

An index can only refer to the columns owned by the same database schema.

Semantics

15.7 ColumnSet Class Diagram

The ColumnSet class diagram provides basic meta-model constructs to define the tables and views of relational databases as collections of columns. The class diagram shown in Figure 15.5 captures these classes and their relations.

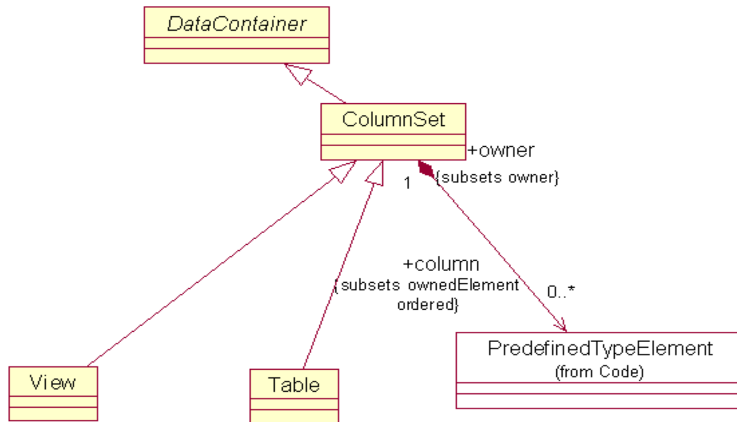

Figure 15.5 - ColumnSet Class Diagram

15.7.1 ColumnSet

The ColumnSet class represents tables and views as collections of columns. Columns are modeled as TypeElements.

Superclass

DataContainer

Associations

column :PredefinedTypeElement[0..*]	Individual columns owned by this ColumnSet are represented as predefined type elements

Semantics

15.7.2 Table Class

A Table is a specific subclass of ColumnSet class that represents tables of relational databases.

Superclass

ColumnSet

Constraints

Semantics

15.7.3 View Class

A View class is a specific subclass of the ColumnSet class that represents Views of relational databases.

Superclass

ColumnSet

Constraints

Semantics

15.8 RecordData Class Diagram

The RecordData class diagram provides basic meta-model constructs to define the indexed files and other kinds of file storage.Usually there is no explicit DataDefinition Language for indexed files. Records are defined by datatypes of the programming language that is selected to work with the indexed file. KDM provides explicit modeling elements. This allows representing the structure of an index file independent on the programming language as well as to precisely define indices of the index files.

The class diagram shown in Figure 15.6 captures these classes and their relations.

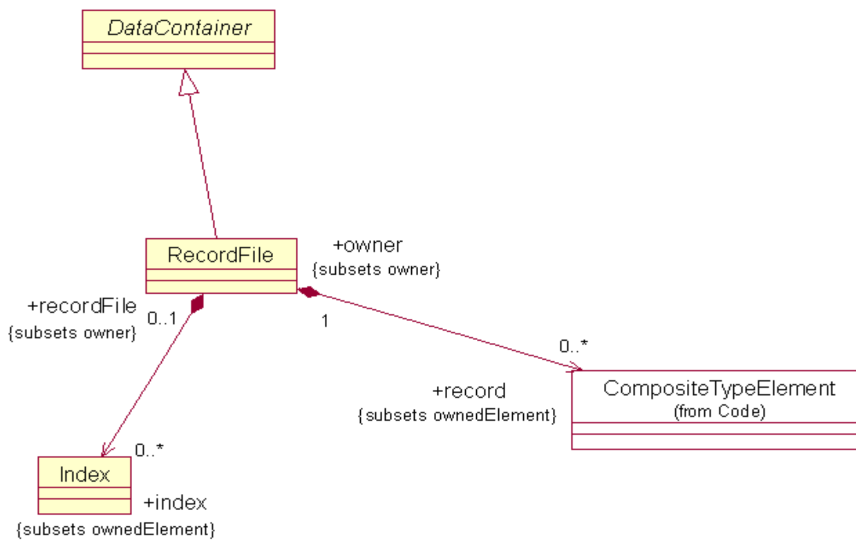

Figure 15.6 - RecordData Class Diagram

15.8.1 RecordFile Class

The RecordFile class represents indexed files.

Superclass

DataContainer

Associations

record :CompositeTypeElement[0..*]	Individual columns owned by this ColumnSet are represented as predefined type elements.
index:Index[0..*]	Indices that are defined for this indexed file.

Constraints

An index can only refer to the elements of the record owned by the same file.

Semantics

15.9 XMLData Class Diagram

The XMLData class diagram provides basic meta-model constructs to define the XML files that can be used by enterprise applications for persistent storage or as an exchange mechanism between components. The class diagram shown in Figure 15.7 captures these classes and their relations.

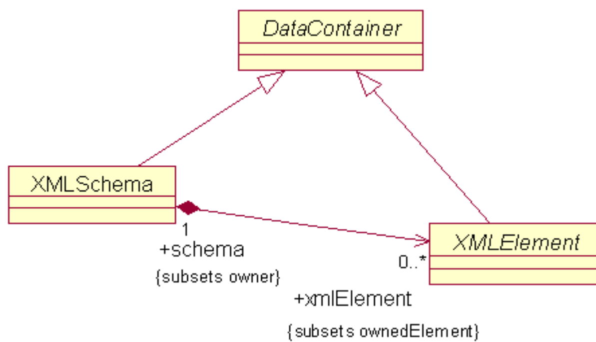

Figure 15.7 - XMLData Class Diagram

15.9.1 XMLSchema

The XMLSchema class represents the top level container for a KDM metamodel of an XML document.

Superclass

DataContainer

Associations

xmlElement :XMLElement[0..*]	Individual XML element owned by this Schema

Semantics

15.10 XMLElements Class Diagram

The XMLElements class diagram defines basic meta-model constructs to represent XML elements. The class diagram shown in Figure 15.8 captures these classes and their relations.

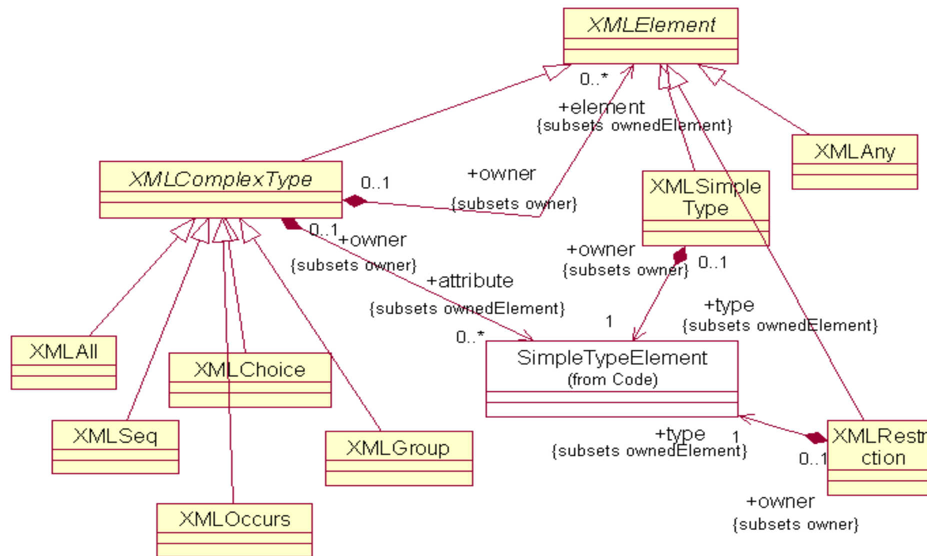

Figure 15.8 - XMLElements Class Diagram

15.10.1 XMLSimpleType

The XMLSimpleType class represents Simple Types of an XML schema definition.

Superclass

XMLElement

Associations

type :SimpleTypeElement[1]	Type of the XML element as Code::SimpleTypeElement

Semantics

15.10.2 XMLRestriction

The XMLRestriction class represents Simple Types with restrictions.

Superclass

XMLElement

Associations

type :SimpleTypeElement[1]	Type of the XML element as Code::SimpleTypeElement

Semantics

15.10.3 XMLComplexType

The XMLComplexType class represents Complex Types of an XML schema definition. XSD indicators are modeled as subclasses of XMLComplexType.

Superclass

XMLElement

Associations

element :XMLElement[0..*]	Owned XML elements
attribute:SimpleTypeElement[0..*]	Owned XML attributes

Semantics

15.10.4 XMLAll Class

An XMLAll class is a specific subclass of the XMLComplexType class that represents complex types with the “all” order indicator.

Superclass

XMLComplexType

Constraints

Semantics

15.10.5 XMLSeq Class

An XMLSeq class is a specific subclass of the XMLComplexType class that represents complex types with the “sequence” order indicator.

Superclass

XMLComplexType

Constraints

Semantics

15.10.6 XMLChoice Class

An XMLChoice class is a specific subclass of the XMLComplexType class that represents complex types with the “choice” order indicator.

Superclass

XMLComplexType

Constraints

Semantics

15.10.7 XMLOccurs Class

An XMLOccurs class is a specific subclass of the XMLComplexType class that represents complex types with the "occurs" occurence indicator.

Superclass

XMLComplexType

Constraints

Semantics

15.10.8 XMLGroup Class

An XMLGroup class is a specific subclass of the XMLComplexType class that represents complex types with the "group" group indicator.

Superclass

XMLComplexType

Constraints

Semantics

15.10.9 XMLAny Class

An XMLAny class is a specific subclass of the XMLElement class that represents the XML "any" type extension mechanism of XML Schemas.

Superclass

XMLElement

Constraints

Semantics

15.11 ProgramElements Class Diagram

The ProgramElements class diagram provides basic meta-model constructs to define the various program elements related to persistent data. The class diagram shown in Figure 15.9 captures these classes and their relations.

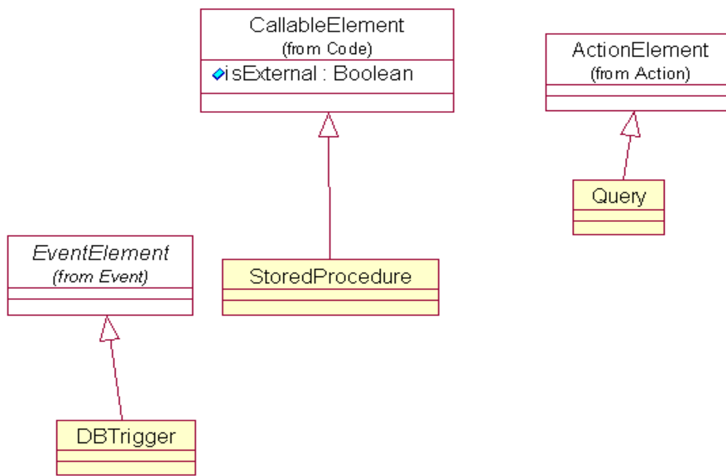

Figure 15.9 - ProgramElements Class Diagram

15.11.1 StoredProcedure Class

The StoredProcedure class extends the Code::CallableElement to represent stored procedures.

Superclass

CallableElement

Semantics

15.11.2 Query Class

The Query class extends the Action::ActionElement class to represent database queries.

Superclass

ActionElement

Semantics

15.11.3 DBTrigger Class

The DBTrigger class extends the Event::EventElement class to represent the database triggers.

Superclass

EventElements

Semantics

15.12 Key Relations class diagram

Figure 15.10 depicts the key relations within the Data Package. A Key is a way to access data without reading through an entire data structure sequentially.

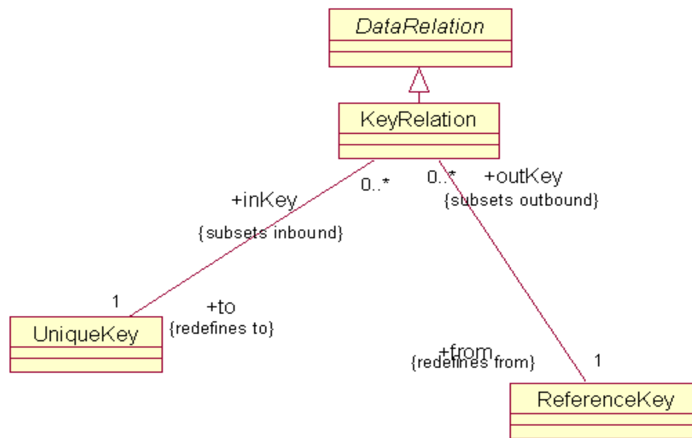

Figure 15.10 - KeyRelations Class Diagram

A KeyRelation class associates a UniqueKey in one data container, which means that there is one and only one key value for that data, with a ReferenceKey in another container.

15.12.1 UniqueKey Class (additional properties)

Associations

inKey : KeyRelationship[0..*]

Constraints

Semantics

15.12.2 ReferenceKey Class (additional properties)

Associations

outKey : KeyRelationship[0..*]

Constraints

Semantics

15.12.3 KeyRelationship Class

A KeyRelationship is a way to associate ReferenceKey with the corresponding UniqueKey. A KeyRelationship is a type of KDMRelationship.

Superclass

DataRelationship

Associations

from : ReferenceKey[1]

to: UniqueKey[1]

Constraints

Semantics

16 Structure Package

16.1 Overview

Structure package defines constructs for defining the high level abstraction of the organization of a software system. The Structure model constructs specify how the software's divisions and subdivisions down to the modules defined in the Code Package.

The form of the system may be presented as a single form or a set of layers components, subsystems, or packages. The reach of this representation extends from a uniform architecture to entire family of module-sharing subsystems.

The Structure model is a collection of StructuralElement instances. A StructuralElement is either a StructuralGroup or a Package.

Packages are the leaf elements of the Structure model, representing a division of a system's Code Modules into discrete, non-overlapping parts. An undifferentiated architecture is represented by a single Package.

StructuralGroup recursively gathers StructuralElements to portray various architectural divisions. The Software System subclass of StructuralGroup supplies a gathering point for all the system's packages directly or indirectly through other StructuralGroups. The packages may be further separated into Subsystems, Layers, and Components.

16.2 Organization of the Structure Package

The Structure package is a collection of classes and associations that are described together because they provide meta-model constructs for defining a software system's architectural organization.

The Structure package depends on the following packages:

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Core

16.3 StructureInheritances Class Diagram

The StructureInheritances class diagram shown in Figure 16.1 depicts how various data classes are types of Core KDM classes. Each of the Structure Package classes within this diagram inherits certain properties from KDM classes defined within the KDM Core Package.

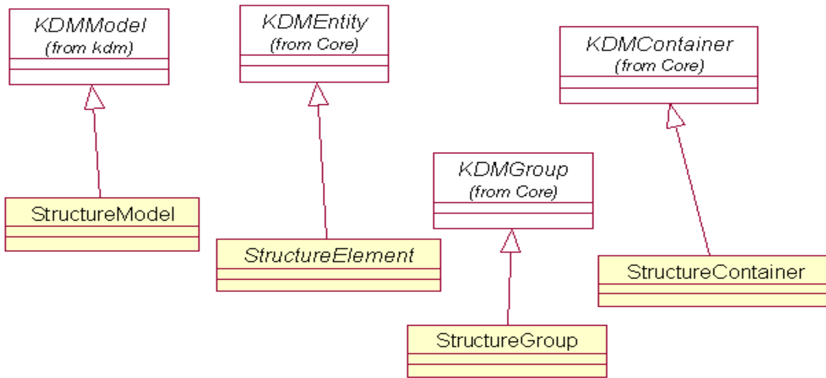

Figure 16.1 - StructureInheritances Class Diagram

16.4 StructureModel Class Diagram

The StructureModel class diagram collects together classes and associations of the Structure package. They provide basic meta-model constructs to define. The class diagram shown in Figure 16.2 captures these classes and their relations.

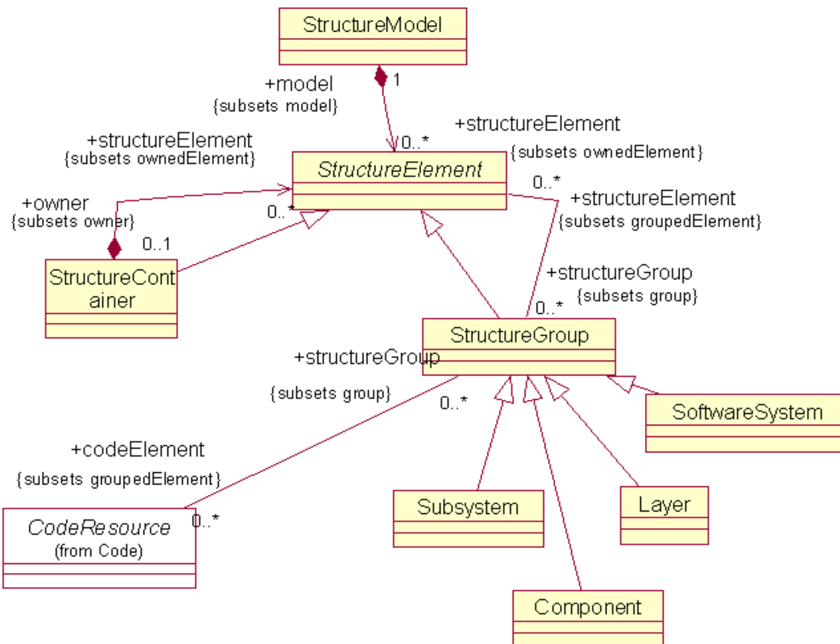

Figure 16.2 - StructureModel Class Diagram

16.4.1 StructureModel Class

The StructureModel is a specific KDM model that represents the logical organization of a software system and owns all of the system's StructuralElements.

Superclass

KDMModel

Associations

structureElement:StructureElement[0..*]

Constraints

Semantics

16.4.2 StructureElement Class

The StructureElement represents an architectural part, which directly or indirectly presents an organization of the system's code modules.

Superclass

KDMEntity

Associations

structureGroup:StructureGroup[0..*]

Constraints

Semantics

16.4.3 StructureGroup Class

The StructureGroup class facilitates a hierarchy or grouping of organization parts within the Structure model.

Superclass

KDMGroup

StructureElement

Associations

codeElement:Code::CodeResource[0..*]

StructureElement:StructureElement[0..*]

Constraints

Semantics

16.4.4 StructureContainer Class

The StructureContainer class facilitates a hierarchy or grouping of organization parts within the Structure model.

Superclass

KDMContainer

StructureElement

Associations

structureElement:Code::StructureElement[0..*]

Constraints

Semantics

16.4.5 Subsystem Class

The Subsystem collects the architectural parts of a software subsystem. The parts may be any other StructuralElement.

Superclass

StructureGroup

Semantics

16.4.6 Layer Class

The Layer collects the architectural parts of a software subsystem to represent a software layer. The parts may be any other StructuralElement.

Superclass

StructureGroup

Semantics

16.4.7 Component Class

The Component represents a collection, directly or indirectly, of code resources, which comprises an architectural component.

Superclass

StructureGroup

Semantics

16.4.8 SoftwareSystem Class

The SoftwareSystem represents the entire system organization. It may contain subsystem or other StructureElements.

Superclass

StructureGroup

Semantics

16.4.9 CodeResource (additional properties)

Associations

structureGroup:StructureGroup[0..*]

Constraints

Semantics

17 Event Package

17.1 Overview

The Event package provides a way for KDM to represent information about the behavior of applications, in particular about the ways in which systems and user interfaces can generate and respond to events and messages.

17.2 Organization of the Event Package

The Event package is a collection of classes and associations that are described together because they provide meta-model constructs for defining the ways in which events and messages are generated and handled by existing systems.

The Event package depends on the following packages:

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Core

17.3 EventInheritances Class Diagram

The EventInheritances class diagram defines how classes of the Event package inherit core meta-model classes from KDM Core package. The classes and associations that make up the EventInherita nces diagram are shown in Figure 17.1.

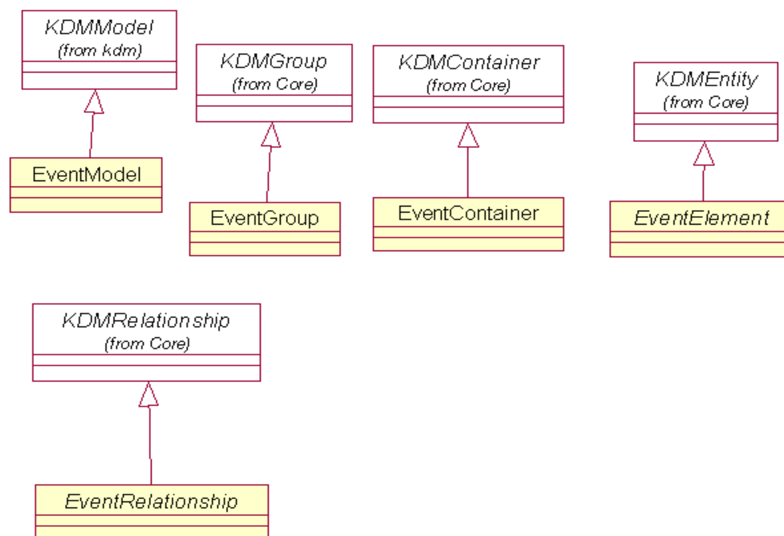

Figure 17.1 - EventInheritances Class Diagram

17.4 EventModel Class Diagram

The EventModel class diagram collects together classes and associations of the Event package. They provide basic meta-model constructs to define event model and event elements.

The class diagram shown in Figure 17.2 captures these classes and their relations.

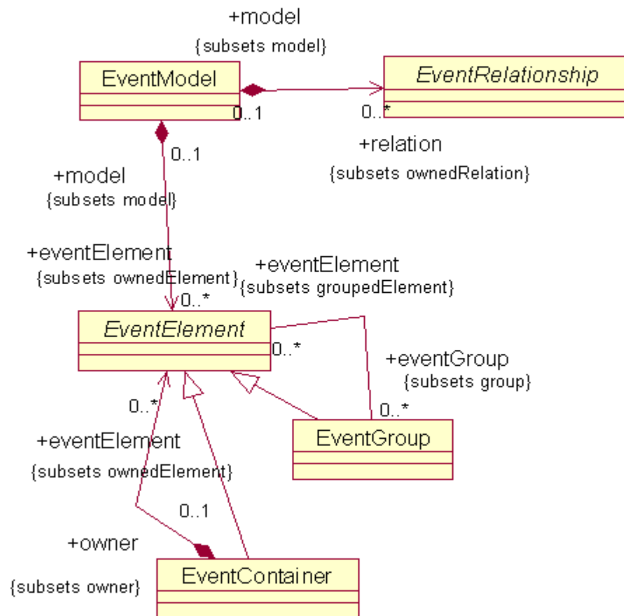

Figure 17.2 - EventModel Class Diagram

17.4.1 EventModel Class

The EventModel is a specific KDM model that represents entities and relations describing events and responses to events in an enterprise application.

Superclass

KDMModel

Associations

eventElement:EventElement[0..*]

Relation:EventRelation[0..*]

Constraints

Semantics

17.4.2 EventElement Class (abstract)

The EventElement represents elementary events such as Trigger instances, or compound events such as EventGroup and EventContainer instances.

Superclass

KDMEntity

Associations

eventGroup:EventGroup[0..*]	The set of EventGroups to which the current EventElement belongs.

Constraints

Semantics

17.4.3 EventGroup Class

The EventGroup provides a means to group EventElements in a structure orthogonal to the containment relationship captured by EventContainer.

Superclass

KDMGroup

EventElement

Associations

eventElement:EventElement[0..*]	

Constraints

Semantics

17.4.4 EventContainer Class

The EventContainer represents the ownership hierarchy of events.

Superclass

KDMContainer

EventElement

Associations

eventElement:EventElement[0..*]	

Constraints

Semantics

17.5 EventElements Class Diagram

The EventElements class diagram defines specific event elements. The class diagram shown in Figure 17.3 captures these classes and their relations.

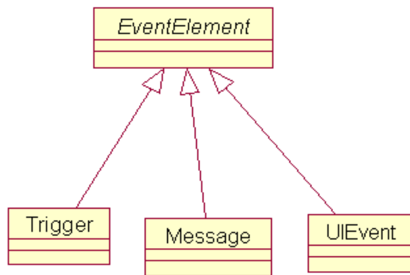

Figure 17.3 - EventElements Class Diagram

17.5.1 Trigger Class

The Trigger is the generic EventElement that can be instantiated in KDM instances.

Superclass

EventElement

Constraints

Semantics

17.5.2 Message Class

The Message is the specific EventElement related to asynchronous message-passing communication mechanisms.

Superclass

EventElement

Constraints

Semantics

17.5.3 UIEvent Class

The UIEvent is the specific EventElement related to the events of the user interfaces.

Superclass

EventElement

Constraints

Semantics

17.6 EventRelations Class Diagram

The EventRelations class diagram defines basic KDM relations between events and other KDM metamodel elements. The class diagram shown in Figure 17.4 captures these classes and their relations.

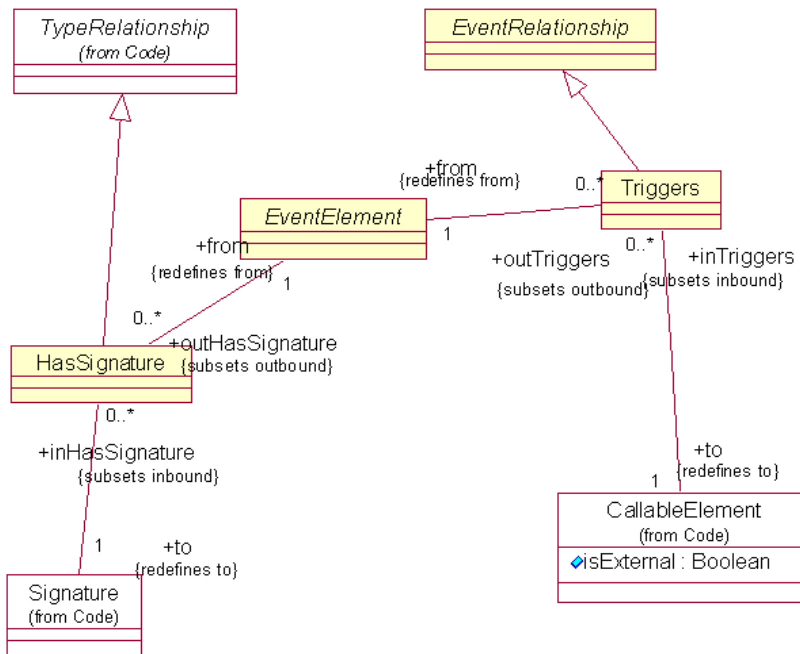

Figure 17.4 - EventRelations Class Diagram

17.6.1 EventRelationship Class (abstract)

The EventRelationship is the superclass of associations within the Event model, which are modeled as classes. Its subclass is Triggers. This is a generic meta-model element for representing various relations involving events.

Superclass

KDMRelationship

Semantics

17.6.2 EventElement Class (additional properties)

Associations

outHasSignature:HasSignature[0..*]	The signature of the trigger
outTriggers:Triggers[0..*]	The set of outbound Triggers relations

Constraints

Semantics

17.6.3 Triggers Class

The Triggers represents the relationship between an EventElement and the CallableElement it causes to be invoked.

Superclass

EventRelationship

Associations

from:EventElement[1]

to:CallableElement[1]

Constraints

Semantics

17.6.4 CallableElement (additional properties)

Associations

inTriggers:Triggers[0..*]

Constraints

Semantics

18 UI Package

18.1 Overview

The UI package provides a way for KDM to represent multiple facets of information about user interfaces, including their composition, their sequence of operations, and their relationships to the underlying software systems.

18.2 Organization of the UI Package

The UI package is a collection of classes and associations that are described together because they provide meta-model constructs for defining the content and behavior of user interfaces.

The UI package depends on the following packages:

org.omg::ADM::KDM::Action

org.omg::ADM::KDM::Build

org.omg::ADM::KDM::Code

org.omg::ADM::KDM::Event

org.omg::ADM::KDM::Core

18.3 UIInheritances Class Diagram

The UIInheritances class diagram defines how classes of the UI package subclass core meta-model elements from the KDM Core package. The classes and associations that make up the UIInheritances class diagram are shown in Figure 18.1.

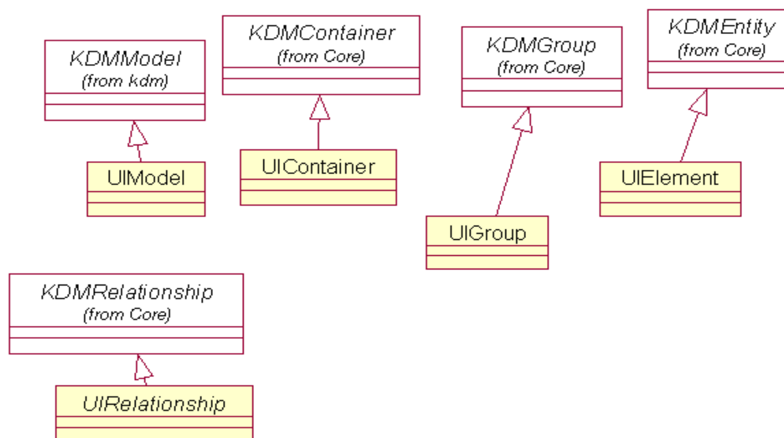

Figure 18.1 - UIInheritances Class Diagram

18.4 UIModel Class Diagram

The UIModel class diagram collects together classes and associations of the UI package. They provide basic meta-model constructs to define a static model of the principal components of a user interface. The class diagram shown in Figure 18.2 captures these classes and their relations.

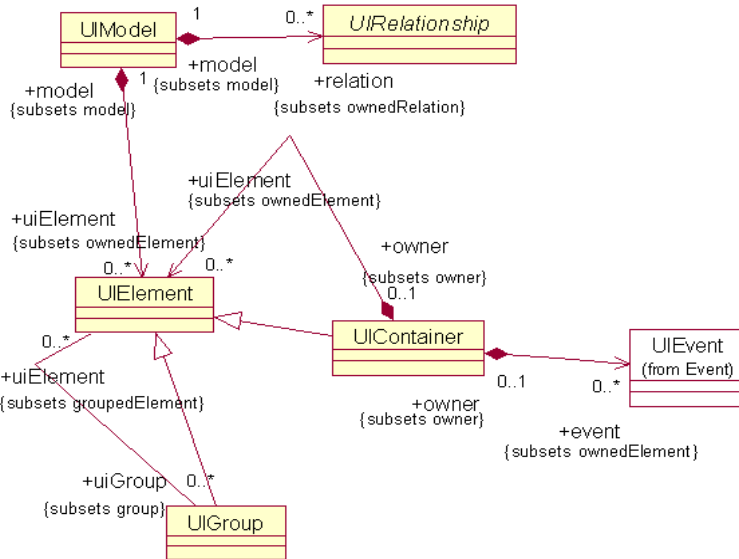

Figure 18.2 - UIModel Class Diagram

18.4.1 UIModel Class

The UIModel is the specific KDM model that represents system's user interface.

Superclass

KDMModel

Associations

uiElement:UIElement[0..*]

relation:UIRelationship[0..*]

Constraints

Semantics

18.4.2 UIElement Class

The UIElement is the superclass of DisplayUnit and UIContainer. As such, it is the class that represents both compound and elementary items in a model of a system's user interface.

Superclass

KDMEntity

Associations

uiGroup:UIGroup[0..1]

Constraints

Semantics

18.4.3 UIContainer Class

The UIContainer is the superclass of Display. It represents a composite of instances of UIElement and Trigger – the information that may be presented on a Screen or Report, and the events that may be generated by a Screen.

Superclass

KDMContainer

UIElement

Associations

uiElement:UIElement[0..*]

event:Event::UIEvent[0..*]

Constraints

Semantics

18.5 Display Class Diagram

The Display class diagram defines several specific KDM containers that own collections of user interface elements. The class diagram shown in Figure 18.3 captures these classes and their relations.

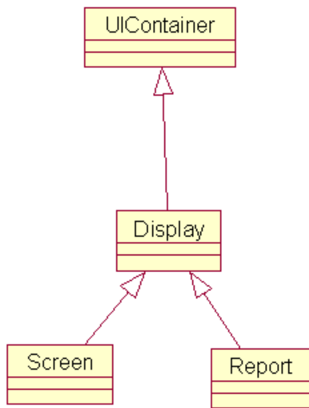

Figure 18.3 - Display Class Diagram

18.5.1 Display Class

The Display is the superclass of Screen and Report. It represents a compound unit of display.

Superclass

UIContainer

Semantics

18.5.2 Screen Class

The Screen is a compound unit of display, such as a Web page or character-mode terminal that is used to present and capture information. The screen may be composed of multiple instances of UIElement and its subclasses.

Superclass

Display

Semantics

18.5.3 Report Class

The Report is a compound unit of display, such as a printed report, that is used to present information. The report may be composed of multiple instances of UIElement and its subclasses.

Superclass

Display

Semantics

18.6 DisplayUnits Class Diagram

The DisplayUnits class diagram provides basic meta-model constructs to define the binding between elements of a display and their content. The class diagram shown in Figure 18.4 captures these classes and their relations.

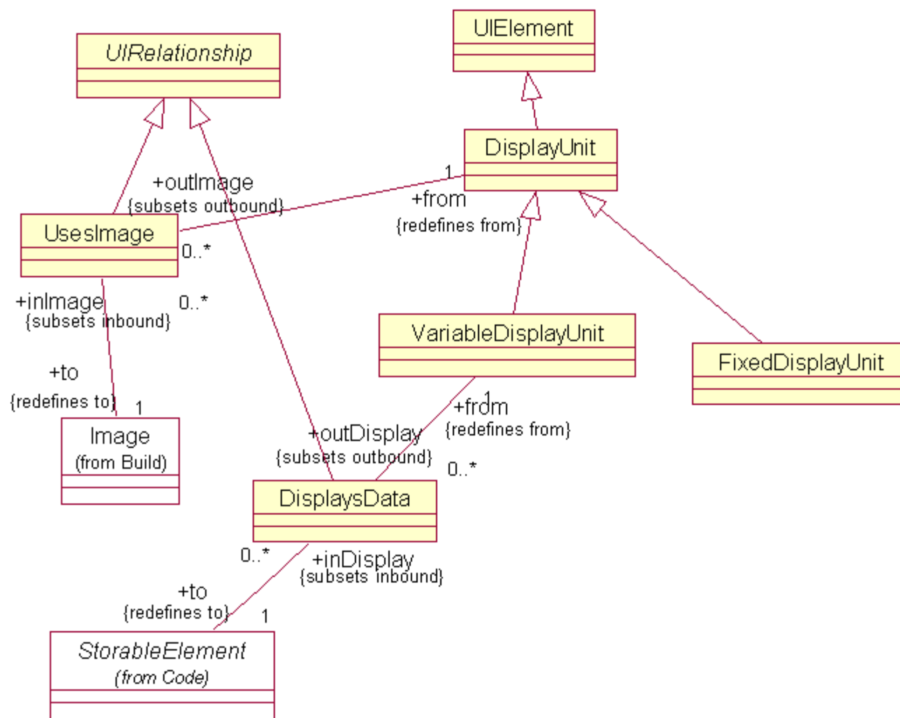

Figure 18.4 - DisplayUnits Class Diagram

18.6.1 DisplayUnit Class

The DisplayUnit is an elementary unit of display, such as a control on a form, a text field on a character-mode terminal, or a field printed on a report. It has two subclasses: FixedDisplayUnit, which is for static content, and VariableDisplayUnit, which is for dynamic content.

Superclass

UIElement

Associations

outImage:UsesImage[0..*]

Constraints

Semantics

18.6.2 FixedDisplayUnit Class

The FixedDisplayUnit represents an element of a user interface that displays static content.

Superclass

DisplayUnit

Semantics

18.6.3 VariableDisplayUnit Class

The VariableDisplayUnit represents an element of a user interface, which displays dynamic content, which is obtained from an instance of DataInterface, which is associated with the VariableDisplayUnit by an instance of the DisplaysData relationship.

Superclass

DisplayUnit

Associations

outDisplay:DisplaysData[0..*]

Constraints

Semantics

18.6.4 UIRelationship Class (abstract)

The UIRelationship is the superclass of associations within the UI model that are modeled as classes. Its subclasses include DisplaysData, UsesImage, Displays, UsesLayout, UIFlow, and Renders.

Superclass

KDMRelationship

Semantics

18.6.5 DisplaysData Class

The DisplaysData captures the relationship between a data source – an instance of DataInterface – and its presentation on a user interface – an instance of VariableDisplayUnit.

Superclass

UIRelationship

Associations

from:VariableDisplayUnit[1]

to:Type::StorableElement[1]

Constraints

Semantics

18.6.6 UsesImage Class

The UsesImage captures the relationship between an image file – an instance of Image – and its presentation on a user interface – an instance of DisplayUnit.

Superclass

UIRelationship

Associations

from:DisplayUnit[1]

to:Build::Image[1]

Constraints

Semantics

18.6.7 StorableElement Class (additional properties)

Associations

inDisplay:DisplaysData[0..*]

Constraints

Semantics

18.6.8 Image Class (additional properties)

Associations

inImage:UsesImage[0..*]

Constraints

Semantics

18.7 UIRelations Class Diagram

The UIRelations class diagram defines several KDM relations for the UI package. It provides basic meta-model constructs to define the sequence of display in a user interface, and the mapping between a user interface and the events it may generate.

The class diagram shown in Figure 18.5 captures these classes and their relations.

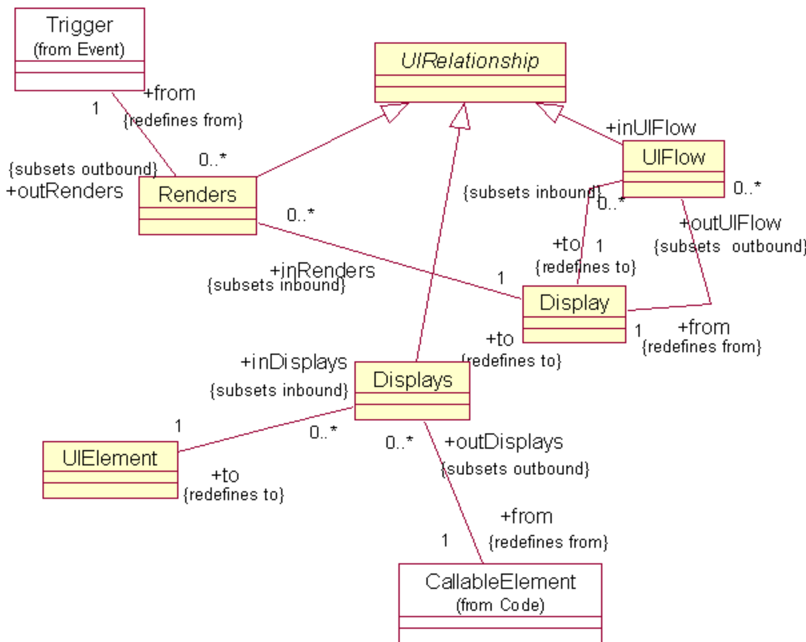

Figure 18.5 - UIRelations Class Diagram

18.7.1 UIFlow Class

The UIFlow relationship class captures the behavior of the user interface as the sequential flow from one instance of Display to another.

Superclass

UIRelationship

Associations

from:Display[1]

to:Display[1]

Constraints

Semantics

18.7.2 Renders Class

The Renders relationship class captures the behavior of the user interface as the collection of Display instances that may be presented as the result of a Trigger.

Superclass

UIRelationship

Associations

from:Event::UIElement[1]

to:Display[1]

Constraints

Semantics

18.7.3 Displays Class

The Displays relationship class represents the relationship between an instance of CallableInterface and the instance of UIElement that is presented on the interface as a result of the execution of the CallableInterface.

Superclass

UIRelationship

Associations

from:CallableElement[1]

to:UIElement[1]

Constraints

Semantics

18.7.4 Display Class (additional properties)

Associations

inRenders:Renders[0..*]

Constraints

Semantics

18.7.5 UIElement Class (additional properties)

Associations

outRenders:Renders[0..*]

inDisplays:Displays[0..*]

Constraints

Semantics

18.7.6 CallableElement (additional properties)

Associations

outDisplays:Displays[0..*]

Constraints

Semantics

18.7.7 UIElement Class (additional properties)

Associations

inDisplays:Displays[0..*]

Constraints

Semantics

18.8 UILayout Class Diagram

The UILayout class diagram collects together classes and associations of the UI package. It provides basic meta-model constructs to define the relationships between user interface content and layout.

The UILayout diagram describes the following types:

- UsesLayout – a class representing the relationship between two instances of Display: One that captures the content of a Screen or Report, and another that captures information on its layout or format.

The class diagram shown in Figure 18.6 captures these classes and their relations.

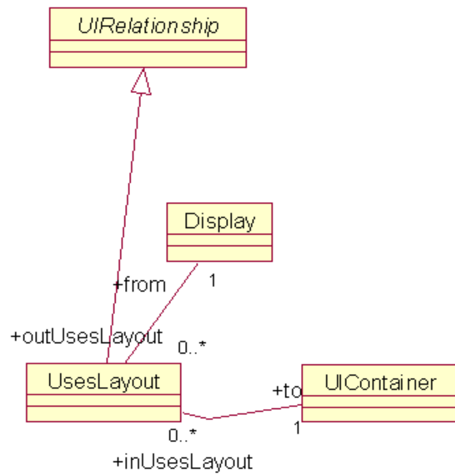

Figure 18.6 - UILayout Class Diagram

18.8.1 UsesLayout Class

The UsesLayout relationship class captures an association between two instances of Display – one that defines the content for a portion of a user interface, and one that defines its layout.

Superclass

UIRelationship

Associations

from:Display[1]

to:UIContainer[1]

Constraints

Semantics

18.8.2 Display Class (additional properties)

Associations

outUsesLayout:UsesLayout[0..*]

Constraints

Semantics

18.8.3 UIContainer Class (additional properties)

Associations

inUsesLayout:UsesLayout[0..*]

Constraints

Semantics

19 Platform Package

19.1 Overview

Platform package provides meta-model constructs for representing operating environments of software systems. Application code is not self-contained as it not only depends on the selected programming language, but also on the selected Runtime platform. Platform elements determine the execution context for the application. Platform package defines meta-model elements that represent common Runtime platform concerns:

- Runtime platform consists of many diverse elements (platform parts).
- Platform provides resources to deployment components.
- Platform provides services that are related to resources.
- Application code invokes services to manage the life-cycle of a resource.
- Control flow between application components is often determined by the platform.
- Platform provides error handling across application components.
- Platform provides integration of application components.

19.2 Organization of the Platform Package

The Platform package is a collection of classes and associations that are described together because they provide meta-model constructs for defining common Runtime platform concerns. The meta-model elements from the Platform package are closely related to the Runtime package, which provides additional meta-model constructs for representing physical deployment of application components and platform elements.

The Platform package depends on the following packages:

org.omg::ADM::KDM::Code
org.omg::ADM::KDM::Runtime
org.omg::ADM::KDM::Core

19.3 PlatformInheritances Class Diagram

The PlatformInheritances class diagram represents inheritances of the meta-modeling elements of the Platform package. The classes and associations that make up the PlatformInheritances diagram are shown in Figure 19.1.

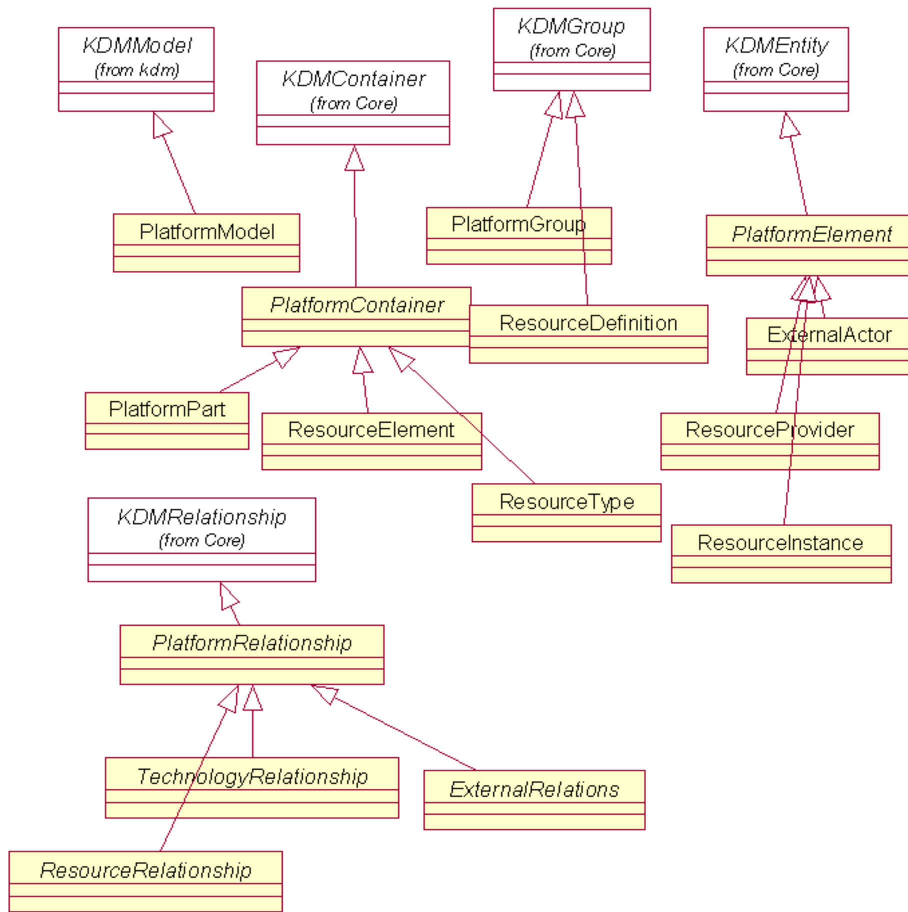

Figure 19.1 - PlatformInheritances Class Diagram

19.4 PlatformModel Class Diagram

The PlatformModel class diagram provides basic meta-model elements that represent platform concerns. The classes and associations that make up the PlatformModel diagram are shown in Figure 19.2.

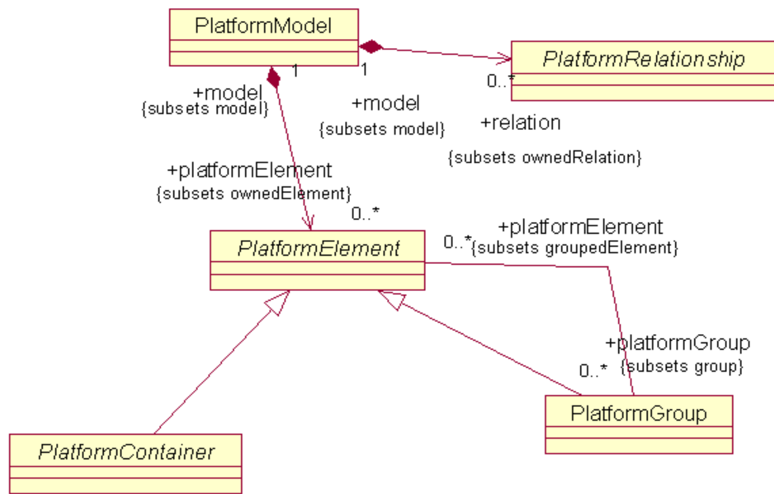

Figure 19.2 - PlatformModel Class Diagram

19.4.1 PlatformModel Class

The PlatformModel is a model element that represents common platform elements and their relationships. Platform model defines one of the architectural views in support of the principle of separation of concerns in KDM models.

In the meta-model, PlatformModel is a subclass of KDMModel.

Superclass

KDMModel

Associations

platformElement:PlatformElement[0..*]

relation:PlatformRelation[0..*]

Constraints

Semantics

19.4.2 PlatformElement Class (abstract)

The PlatformElement is a meta-model element that represents entities of the operating environments of software systems. In the meta-model a PlatformElement is a subclass of KDMEntity.

Superclass

KDMEntity

Associations

platformGroup:PlatformGroup[0..*]	The set of PlatformGroups with which the current element is associated.

Constraints

Semantics

19.4.3 PlatformGroup Class

The PlatformGroup is a meta-model element that provides generic grouping capabilities for PlatformElements. PlatformGroup is associated with a set of PlatformElements. A PlatformElement can be associated with multiple PlatformGroups. In the meta-model PlatformGroup is a subclass of KDMGroup.

A PlatformElement can be associated with multiple PlatformGroups. The association between the PlatformGroup and the PlatformElements that are associated with that group is one directional.

Superclass

KDMGroup

PlatformElement

Associations

platformElement:PlatformElement[0..*]	The set of PlatformElements associated with the current PlatformGroup.

Constraints

Semantics

19.4.4 PlatformContainer Class (abstract)

The PlatformContainer is a meta-model element that provides generic grouping capabilities for PlatformElements. PlatformContainer owns a set of PlatformElements. A PlatformElement can be associated with a single PlatformContainer. In the meta-model PlatformContainer is a subclass of KDMContainer. It is also a subclass of PlatformElement.

A PlatformElement can be owned by at most one PlatformContainer. The optional character of ownership is intended as a convenience to tools, allowing them to create PlatformElements prior to linking them to the owning PlatformContainer. PlatformElements without an owner PlatformContainer also capture the top level element (especially PlatformContainers and PlatformGroups).

PlatformContainer is an abstract class. The actual associations are defined by its subclasses.

Superclass

PlatformElement

KDMContainer

Constraints

Semantics

19.5 PlatformResources Class Diagram

The PlatformResources class diagram defines the meta-model elements to represent platform resources. The classes and associations that make up the PlatformResources diagram are shown in Figure 19.3.

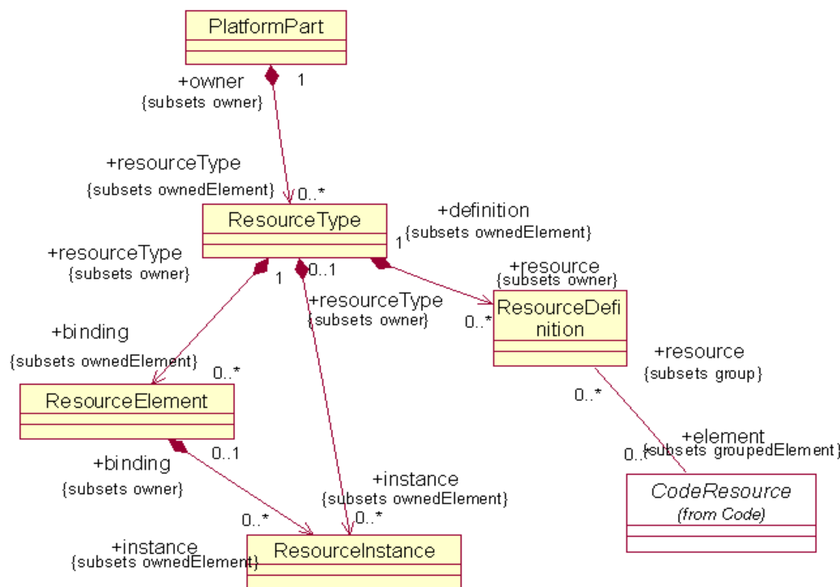

Figure 19.3 - PlatformResources Class Diagram

19.5.1 PlatformPart Class

The PlatformPart is a meta-model element that represents a coherent set of resources that application components can manage and share. Platform parts can be further subdivided into smaller groups of services (resource types). Platform parts are usually grouped into platform stacks. Platform part is an element of the overall platform used by a particular system. Complete Platform is the entire collection of platform parts used by the segments of the system. Platform Part may be associated with logical packages for a particular programming language.

Examples of Platform Parts include UNIX OS File System, UNIX OS process management system, Windows 2000, OS/390, Java (J2SE), Perl language Runtime support, IBM CICS TS, IBM MQSeries, Jakarta Struts, BEA Tuxedo, CORBA, HTTP, TCP/IP, Eclipse, EJB, JMS, Database middleware, Servlets.

In the meta-model PlatformPart is a subclass of PlatformContainer.

Superclass

PlatformContainer

Associations

resourceType:ResourceType[0..*]	The set of ResourceTypes owned (supported) by the target PlatformPar.

Constraints

Semantics

19.5.2 ResourceElement Class

The ResourceElement is a meta-model element that represents an instance of a resource type.

In the meta-model ResourceInstance is a subclass of PlatformContainer. KDM makes distinction between ResourceInstance and ResourceElement. A ResourceElement is a globally known instance of a resource type. A ResourceInstance is a locally known resource instance as seen by a component. The binding between components and ResourceInstances is expected to be resolved within a single model Segment while the binding between ResourceInstances and ResourceElements is expected to be resolved during integration of multiple model segments.

Superclass

PlatformContainer

Associations

instance:ResourceInstance[0..*]	The set of ResourceInstances that are owned by the target ResourceElement (matched to represent the same global resource entity).

Constraints

Semantics

19.5.3 ResourceInstance Class

The ResourceInstance is a meta-model element that represents an instance of a resource type.

In the meta-model ResourceInstance is a subclass of PlatformElement. ResourceInstance is the endpoint of relations. It represents a local view of a certain resource type, as seen by a component. The binding between components and ResourceInstances is expected to be resolved within a single model Segment. In order to support integration of KDM models within KDM, it is important to support partial views. The distinction between ResourceInstance and ResourceElement is as follows:

- A ResourceElement is a globally known instance of a resource type.
- A ResourceInstance is a locally known resource instance. Often, components are integrated by glue-ware, e.g., Jcl scripts, which provide the binding between local resource names.

Separation between ResourceElement and ResourceInstance support matching of instances to elements within KDM. When a ResourceInstance is contained in ResourceElement, it is "matched" to that element. If it is not matched to any ResourceElement, it is unmatched (yet). Therefore it is possible to have a ResourceElement without any ResourceInstances - this represents, e.g., the view of an integration glue-ware, without the actual components.

Superclass

PlatformElement

Constraints

Semantics

19.5.4 ResourceType Class

The ResourceType is a meta-model element that represents platform resource. In the meta-model ResourceType is a subclass of PlatformContainer. The purpose of a platform is to simplify application development by closing the gap between the application domain and the facilities that are available to application programmers. The latter are referred to as platform resources. Examples of resource types include UNIX File, UNIX IO Stream, UNIX socket, UNIX Process, UNIX thread, AWT widget, CICS File, CICS transaction, UNIX semaphore, UNIX shared memory segment, OS/390 VSAM file, JDBC connection, HTTP session, HTTP request, UNIX memory block, CICS commarea, COBOL file.

KDM introduces Platform Resource as an explicit abstraction in order to separate the explicit parts that are written by application programmers from parts that are provided by the platform. The underlying implementation details may be quite complex, for example, marshaled call includes client stubs, skeletons, platform-managers. The type of the Platform Resource denotes the semantics of the resource. KDM only allows distinguishing platform resources with different type names, but the definition of the semantics of a resource type is beyond the scope of KDM.

Superclass

PlatformContainer

Associations

binding:ResourceElement[0..*]	The set of ResourceElements that are owned by the target ResourceType.
instance:ResourceInstance[0..*]	The set of ResourceInstances that are owned by the target ResourceType and not associated with a particular ResourceElement.
resource: ResourceDefinition[0..*]	The set of ResourceDefinitions that provide logical definitions of the resources of the target ResourceType.

Constraints

Semantics

19.5.5 ResourceDefinition Class

The ResourceDefinitions is a meta-model element that represents a collection of code resource associated with a particular platform resource type. In the meta-model ResourceType is a subclass of PlatformGroup.

Superclass

PlatformGroup

Associations

element:CodeResource[0..*]	The set of CodeResources that are owned by the target ResourceType.

Constraints

19.6 ResourceTypes Class Diagram

The ResourceTypes class diagram defines several basic subclasses of ResourceType. The classes and associations that make up the ResourceTypes diagram are shown in Figure 19.4.

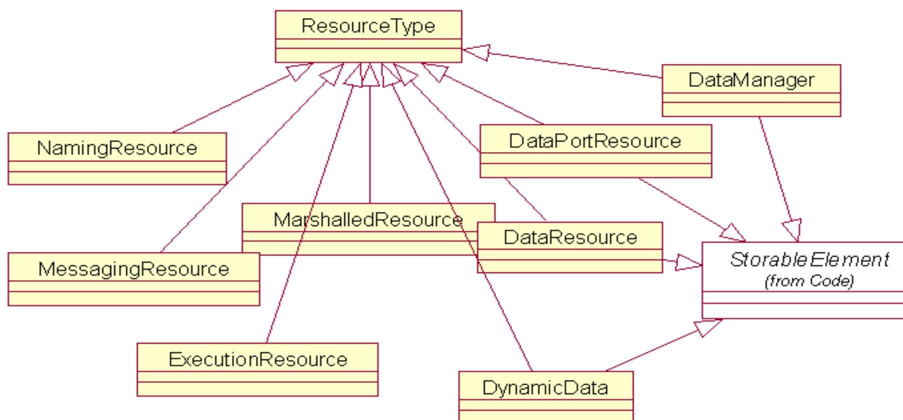

Figure 19.4 - ResourceTypes Class Diagram

19.6.1 NamingResource Class

NamingResource represents platform resources that provide registration and lookup services (e.g., registry). In the meta-model NamingResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

19.6.2 MarshaledResource Class

MarshaledResource represents platform resources that provide intercomponent communication via remote synchronous calls. For example, RPC, CORBA method call, Java remote method invocation. In the meta-model MarshaledResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

19.6.3 MessagingResource Class

MessagingResource represents platform resources that provide intercomponent communication via asynchronous messages (e.g., IBM MQSeries messages).

In the meta-model MessagingResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

19.6.4 DataResource Class

DataResource represents platform resources that provide any non-database related storage. In the meta-model the DataResource class is a subclass of ResourceType. It also implements the DataInterface so that this class can be the endpoint of Data relations.

Superclass

StorableElement

ResourceType

Semantics

19.6.5 ExecutionResource Class

ExecutionResource represents dynamic Runtime elements (e.g., process or thread). In the meta-model ExecutionResource is a subclass of ResourceType.

Superclass

ResourceType

Semantics

19.6.6 DataPortResource Class

DataPortResource represents a platform resource that provides communication between components of the system and external entities in the operating environment of the system. In the meta-model the DataPortResource class is a subclass of ResourceType. It also implements the DataInterface so that this class can be the endpoint of Data relations.

Superclass

StorableElement

ResourceType

Semantics

19.6.7 DynamicData Class

DynamicData represents dynamic data elements (e.g., string buffers). In the meta-model the DynamicData class is a subclass of ResourceType. It also implements the DataInterface so that this class can be the endpoint of Data relations.

Superclass

StorableElement

ResourceType

Semantics

19.6.8 DataManager Class

DataManager represents a database management system. DataManager is associated with particular data elements that represent the data description of the data managed by the data manager. In the meta-model the DataManager class is a subclass of ResourceType. It also implements the DataInterface so that this class can be the endpoint of Data relations.

Superclass

StorableElement

ResourceType

Semantics

19.7 ExternalActors Class Diagram

The ExternalActors class diagram defines the environment of the system. It introduces meta-model elements for representing entities outside of the system and relations between external entities and the system.

The classes and associations that make up the ExternalActors diagram are shown in Figure 19.5.

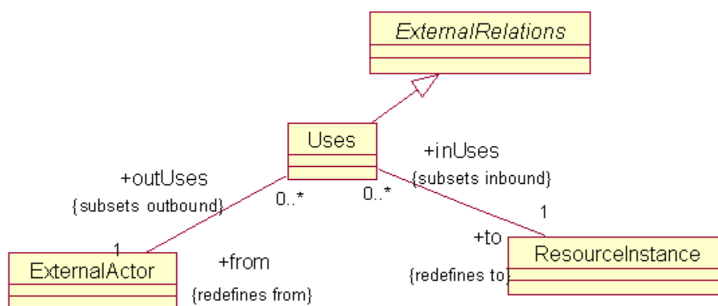

Figure 19.5 - ExternalActors Class Diagram

19.7.1 ExternalActor Class

ExternalActor is a meta-model element that represents entities outside of the boundary of the software system being modeled. For example, external actor can be a user of the system. External actors interact with the software system.

In the meta-model ExternalActor is a PlatformElement. Semantics of ExternalActors is outside of the scope of KDM.

Superclass

PlatformElement

Associations

outUses:Uses[0..*]	The set of Uses relationships that originate from the target ExternalActor.

Constraints

Semantics

19.7.2 ExternalRelations Class (abstract)

ExternalRelations is a generic meta-model element that represents various relationships between the software system and its environment. In the meta-model the ExternalRelations class is a subclass of KDMRelationship. This class is an extension point. In KDM there is only one subclass of ExternalRelations – the class UsesActor.

Superclass

KDMRelationship

Semantics

19.7.3 Uses Class

Uses class is a meta-model element that represents associations between external actors outside of the software system and the entities inside the system. External actors always interact through the Runtime platform. In the meta-model the Uses class is a subclass of ExternalRelations class.

Superclass

ExternalRelations

Associations

from:ExternalActor[1]	The ExternalActor which is the origin of the target Uses relation (the from-endpoint).
to:ResourceInstance[1]	The ResourceInstance which is the target of the Uses relation (the to-endpoint).

Constraints

Semantics

19.7.4 ResourceInstance (additional properties)

Associations

inUses:Uses[0..*]	The inbound Uses relationships for which the target ResourceInstance is the to-endpoint

Constraints

Semantics

19.8 PlatformInterfaces Class Diagram

The PlatformInterfaces class diagram defines meta-model associations between platform elements and interfaces. The classes and associations that make up the PlatformInterfaces diagram are shown in Figure 19.6.

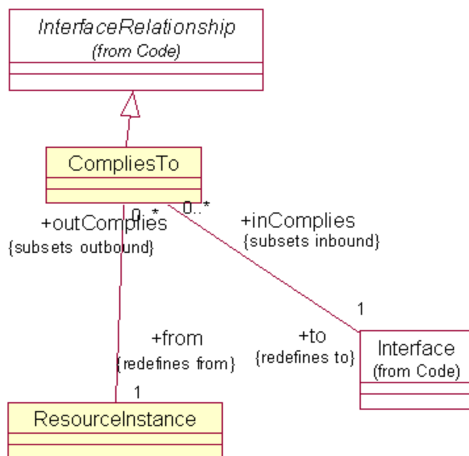

Figure 19.6 - PlatformInterfaces Class Diagram

19.8.1 CompliesTo Class

CompliesTo class defines semantic relationship between a resource instance and an Interface class from Code package. The CompliesTo relationship represents the fact that a certain resource instance complies with a certain interface.

In the meta-model CompliesTo class is a subclass of InterfaceRelation, defined in the Code package.

Superclass

InterfaceRelationship

Associations

from:ResourceInstance[1]	The source ResourceInstance
to:Code::Interface[1]	The target Interface class

Constraints

Semantics

19.8.2 ResourceInstance (additional properties)

Associations

outComplies:CompliesTo[0..*]	The outbound CompliesTo relationships for which the target ResourceInstance is the from-endpoint.

Constraints

Semantics

19.8.3 Interface (additional properties)

Associations

inComplies:CompliesTo[0..*]

Constraints

Semantics

19.9 PlatformRelations Class Diagram

The PlatformRelations class diagram defines associations between ResourceInstances and CodeResource (the so-called bindings of ResourceInstances to CodeResources).

The classes and associations that make up the PlatformRelations diagram are shown in Figure 19.7.

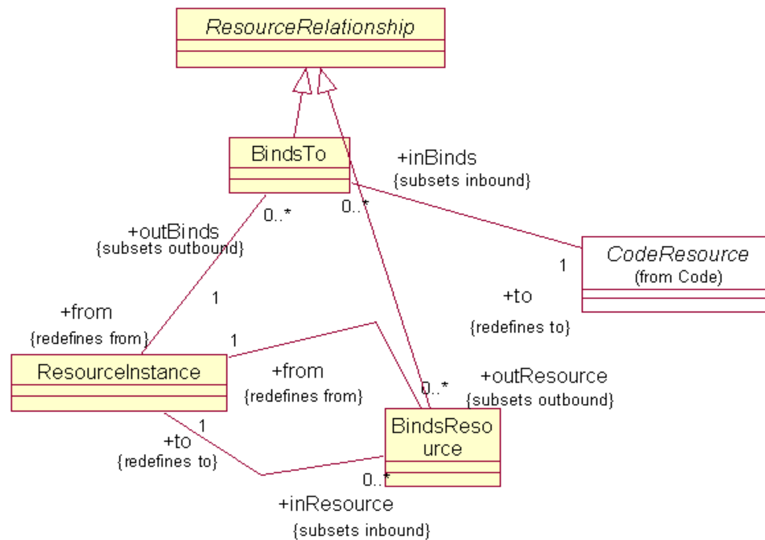

Figure 19.7 - PlatformRelations Class Diagram

19.9.1 BindsTo Class

BindsTo defines a semantic relationship between a ResourceInstance and a CodeResource. The CodeResource represents a callback (the so called “reversed” control flow) associated with the target ResourceInstance. Major platform parts support componentization by “reversing” some of the control flows. “Reversed” control flows reduce coupling between components (but not necessarily eliminate it). Deployment components are usually *plugins* into the platform code. Initially, control belongs to the platform code. Platform code *activates* deployment components through various kinds of callback mechanisms, etc. A BindsTo relation represents these platform-specific activations. There are several activation models:

- Interruptible (processes, threads) vs. run-to-completion (application frameworks)
- Synchronous vs. asynchronous

Knowledge of platform-specific activations is essential for understanding a software system as it provides execution context for understanding flow of control through the software system.

Superclass

ResourceRelationship

Associations

from:ResourceInstance[1]	The ResourceInstance which is the source of the relationship (the from-endpoint).
to:CodeResource[1]	The CodeResource which is the target of the relationship (the to-endpoint).

Constraints

Semantics

19.9.2 ResourceInstance (additional properties)

Associations

outBinds:BindsTo[0..*]	The outbound BindsTo relationships for which the target ResourceInstance is the from-endpoint.

Constraints

Semantics

19.9.3 CodeResource (additional properties)

Associations

inBinds:BindsTo[0..*]

Constraints

Semantics

19.10 ProvisioningRelations Class Diagram

The ProvisioningRelations class diagram defines meta-model elements that represent the physical elements of the Runtime platform that provide certain services and manage resources.

The classes and associations that make up the ProvisioningRelations diagram are shown in Figure 19.8.

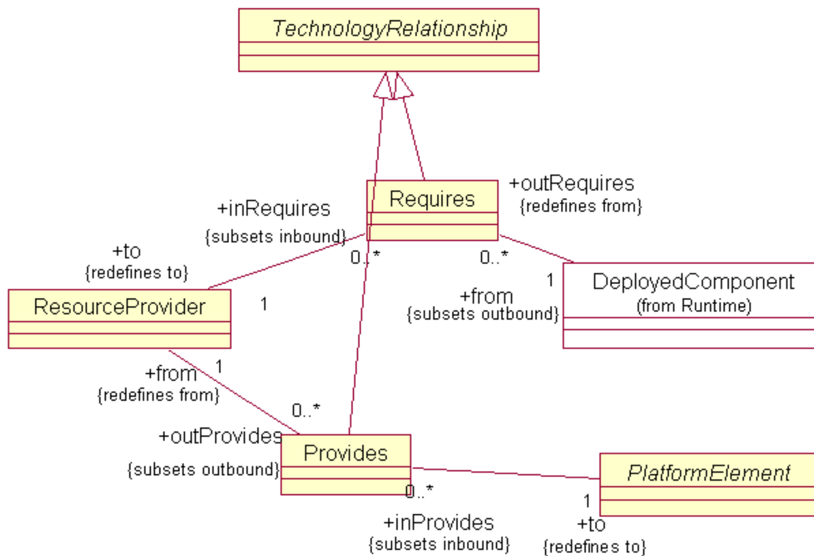

Figure 19.8 - ProvisioningRelations Class Diagram

19.10.1 PlatformProvider Class

PlatformProvider is a meta-model element that represents physical entities of the Runtime platform that provide certain services and manage resources.

Superclass

KDMEntity

Associations

outProvides:Provides[0..*]	The set of Provides relationships which originate at the target PlatformProvider.
inRequires:Requires[0..*]	The set of Requires relationships for which the target PlatformProvider is the to-endpoint.

Constraints

Semantics

19.10.2 TechnologyRelationship Class (abstract)

TechnologyRelationship is a generic meta-model element that defines semantic associations related to provisioning of platform services to applications.

In the meta-model the TechnologyRelationship is a subclass of KDMRelationship. It is further subclassed by concrete KDM technology relationships. It provides an extension point for lightweight extensions.

Superclass

KDMRelationship

Semantics

19.10.3 Requires Class

Requires defines semantic relationship between a DeployedComponent and a PlatformProvider.

In the meta-model Requires is a subclass of TechnologyRelationship.

Superclass

TechnologyRelationship

Associations

from:DeployedComponent[1]	The DeployedComponent which is the source of the relationship (the from-endpoint).
to:PlatformProvider[1]	The PlatformProvider which is the target of the relationship (the to-endpoint) .

Constraints

Semantics

19.10.4 Provides Class

Provides defines semantic relationship between a PlatformProvider and a PlatfromElement.

In the meta-model Provides is a subclass of TechnologyRelationship.

Superclass

TechnologyRelationship

Associations

from: PlatformProvider [1]	The PlatformProvider which is the source of the relationship (the from-endpoint).
to: PlatformElement [1]	The PlatformElement which is the target of the relationship (the to-endpoint).

Constraints

Semantics

19.10.5 PlatformElement (additional properties)

Associations

inProvides:Provides[0..*]	The set of Provides classes (a subclass of TechnologyRelation) that represent inbound relationships from various PlatformProvider classes for which the target PlatformElement is the to-endpoint. This association is defined in the ProvisioningRelations class diagram of the Platform package.

Constraints

Semantics

19.10.6 DeployedComponent (additional properties)

Associations

outRequires:Requires[0..*]	The set of Requires classes (a subclass of TechnologyRelation) that represent outbound relationships from various DeployedComponent classes for which the target PlatformElement is the to-endpoint. This association is defined in the ProvisioningRelations class diagram of the Platform package.

Constraints

Semantics

19.11 PlatformActions Class Diagram

The PlatformActions class diagram defines meta-model elements that represent specific Runtime actions as the endpoints of certain Runtime and Platform relationships. The elements of this diagram extend ActionElement from the KDM Action package.

The classes and associations that make up the PlatformActions diagram are shown in Figure 19.9.

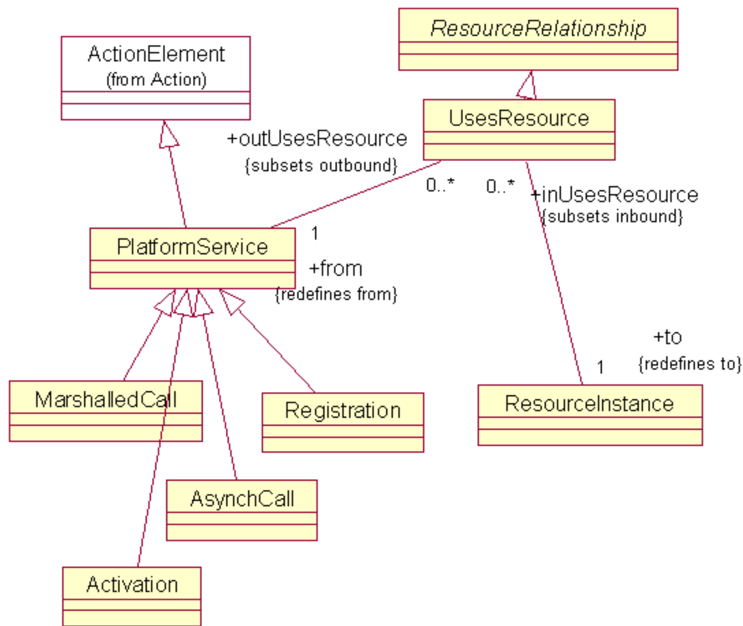

Figure 19.9 - PlatformActions Class Diagram

19.11.1 PlatformService Class

The PlatformService is a generic meta-model element that represents various endpoints of ResourceRelation.

In the meta-model PlatformService is a subclass of ActionElement. This class is an extension point. In KDM PlatformService is subclassed by several specific concrete actions that correspond to specific ResourceRelations.

Currently specific subclasses of PlatformService are not strongly typed because KDM does not provide any subclasses of ResourceRelationship class and no specific subclasses of ResourceInstance that correspond to concrete ResourceType. PlatformService element directly references ResourceRelationship element. Later KDM will introduce more specific subclasses of ResourceInstances and corresponding subclasses of ResourceRelationships, which will allow stronger typing of specific subclasses of PlatformService.

Superclass

ActionElement

Associations

outResource:ResourceRelationship[0..*]	The set of outbound ResourceRelationships.

Constraints

Semantics

19.11.2 MarshaledCall Class

The MarshaledCall is a meta-model element that represents the action endpoint of a “marshaled call” relation to some instance of marshaled resource. Marshaled call is some kind of a remote procedure call, supported by the platform. “Marshaled call” relation abstracts proxies, stubs and the corresponding platform code. Examples of marshaled calls include CORBA method invocations, RMI method calls, RPC, COM method calls, database notifications via callbacks.

In the meta-model MarshaledCall is a subclass of a generic PlatformService element.

Superclass

PlatformService

Semantics

19.11.3 AsynchCall Class

The AsynchCall is a meta-model element that represents the action endpoint of an “asynchronous message passing” relation to some instance of messaging resource. Asynchronous call is some kind of a messaging, supported by the platform. “Asynchronous call” relation abstracts proxies, stubs, and the corresponding platform code. Examples of asynchronous calls include IBM MQSeries messaging, Microsoft MSMQ messaging, BEA Tuxedo.

In the meta-model AsynchCall is a subclass of a generic PlatformService element.

Superclass

PlatformService

Semantics

19.11.4 Registration Class

The Registration is a meta-model element that represents the action endpoint of a “naming service registration and lookup” relation to some instance of naming resource. Registration is some kind of a naming service operation, supported by the platform. Examples of registration include Java registries, Windows registries, CORBA naming service.

In the meta-model Registration is a subclass of a generic PlatformService element.

Superclass

PlatformService

Semantics

19.11.5 Activation Class

The Activation is a meta-model element that represents the action endpoint of an “execution” relation to some instance of executable resource. Execution is some kind of a component invocation operation, supported by the platform. Examples of activation include system() calls on UNIX, script activations.

In the meta-model Activation is a subclass of a generic PlatformService element.

Superclass

PlatformService

Semantics

19.11.6 ResourceRelationship Class (abstract)

The ResourceRelationship is a generic meta-model element that represents various platform-specific relations.

In the meta-model ResourceRelationship is a subclass of KDMRelationship. Currently KDM does not provide any subclasses of ResourceRelationship class as there are no specific subclasses of ResourceInstance that correspond to concrete ResourceTypes which are introduced in the Platform package. ResourceRelationship directly references ResourceInstance element. Later KDM will introduce more specific subclasses of ResourceInstances and corresponding subclasses of ResourceRelationships.

Superclass

KDMRelationship

Associations

from:PlatformService[1]	The PlatformService endpoint
to:ResourceInstance[1]	The ResourceInstance

Constraints

Semantics

19.11.7 ResourceInstance (additional properties)

Associations

inResource:ResourceRelationship[0..*]	The set of inbound ResourceRelationships

Constraints

Semantics

20 Runtime Package

20.1 Overview

The Runtime package provides meta-model constructs for representing the physical aspects of operating environments of 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. Runtime package defines meta-model elements that represent common physical aspects of Runtime platform concerns:

- Runtime platform consists of many diverse elements (platform parts)
- Platform provides resources to deployment components
- Platform provides services that are related to resources
- Application code invokes services to manage the life-cycle of a resource
- Control flow between application components is often determined by the platform
- Platform provides error handling across application components
- Platform provides integration of application components

Additional Runtime concerns involve dynamic structures (instances of some logical entities and their relationships) that emerge at the so-called “run time” of the software system. For example, dynamic entities include processes and threads. Instances of processes and threads can be created dynamically and in many cases relations between the dynamically created instances of processes and threads are essential part of the knowledge of existing system. Pure logical perspective in that case may be insufficient.

When separation of concerns between application code and Runtime platform is considered, it is important to be clear about the so-called *bindings* and various mechanisms to achieve a binding (or delay it).

A binding is a common way of referring to a certain irrevocable implementation decision.

Too much binding is often referred to as “hardcoding”. This often results in systems that are difficult to maintain and reuse. They are often also difficult to understand.

Too little binding leads to dynamic systems, where everything is resolved at run time (as late as possible). This often results in systems that are difficult to understand and error-prone.

Modern platforms excel in ingenious ways to manage binding time. Usually binding is managed at deployment time.

Large number of software development practices is about efficient management of binding time, including portable adaptors, code generation, model-driven architecture.

Efficient management of binding time is often called platform independence.

Binding time

- Generation time binding
- Language & platform design binding
- Versioning time
- Compile time binding, including
  - macro expansion
  - Templates

    • Product line variants defined by conditional compilation
- Link time binding
- Deployment time binding
- Initialization time binding
- Run time

Binding Time	**What**	**Result**
Generation time	Syntax, variant, pattern, mapping, etc.	Generated code
Language & platform design	Syntax, entities and relations, including platform resource types	Source code
Versioning	Module source files	Module version
Compile time	Intra-module relations (def-use)	Module
-- Macro	Syntax, macro to expanded code	Expanded macro (source code)
-- Template	Template parameters	Template instance
-- Product line variant defined by conditional compilation and includes	Conditional compilation, macro, includes, symbolic links	Component Variant
(static) Link time	Intra-component relations within deployable component	Deployed Component
Deployment time	Resource names to resources (using platform-specific configuration files)	Deployed System
Initialization time	Component implementation to component interface; major processes and threads; dynamic linking, dynamic load (using platform-specific configuration files)	System
Run time	User input, object factory, virtual function, function pointer, reflection, instances of processes, instances of objects, instances of data, etc.	Particular Execution Path

20.2 Organization of the Runtime Package

The Runtime package is a collection of classes and associations that are described together because they provide meta-model constructs for defining common physical aspects of Runtime platform concerns. The meta-model elements from the Runtime package are closely related to the Platform package, which provides additional meta-model constructs for representing logical aspects of Runtime platforms.

The Runtime package depends on the following packages:

org.omg::ADM::KDM::Platform

org.omg::ADM::KDM::Structure

org.omg::ADM::KDM::Action

org.omg::ADM::KDM::Core

20.3 RuntimeInheritances Class Diagram

The RuntimeInheritances class diagram collects together classes and associations of the Runtime package. They provide basic meta-model constructs to define The classes and associations that make up RuntimeInheritances diagram are shown in Figure 20.1.

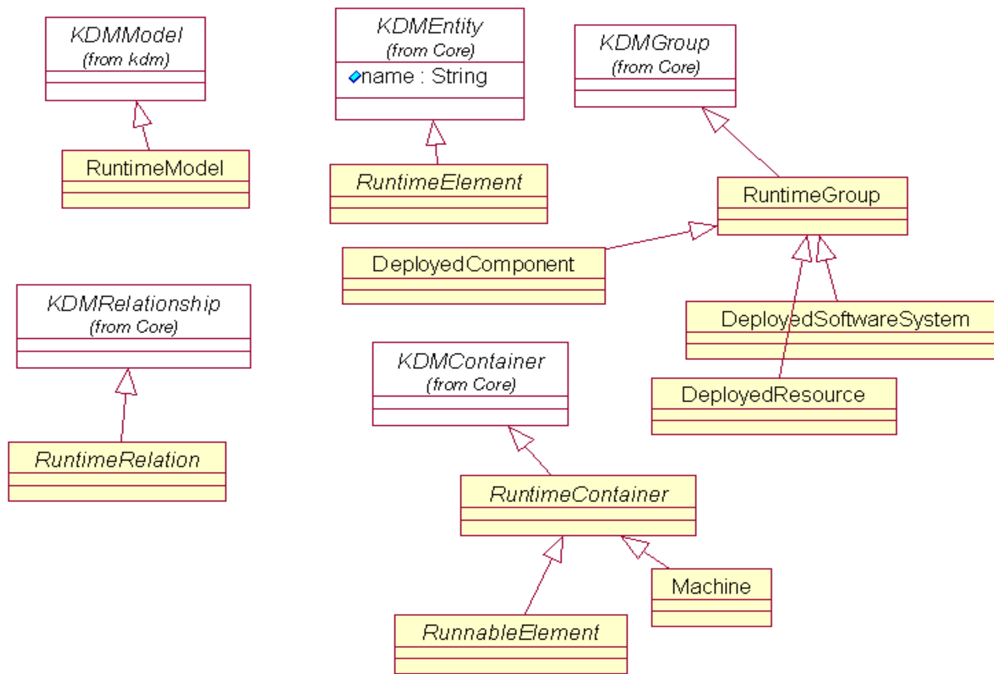

Figure 20.1 - RuntimeInheritances Class Diagram

20.4 RuntimeModel Class Diagram

The RuntimeModel class diagram provides basic meta-model elements that represent physical Runtime concerns. The classes and associations that make up the RuntimeModel diagram are shown in Figure 20.2.

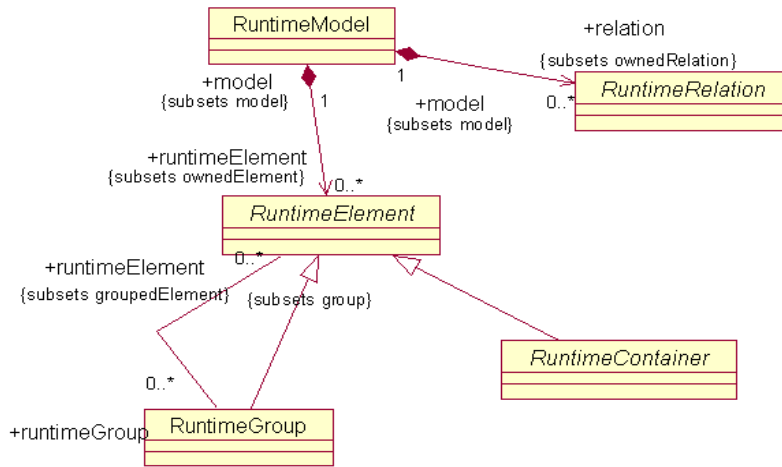

Figure 20.2 - RuntimeModel Class Diagram

20.4.1 RuntimeModel Class

The RuntimeModel is a model element that represents generic Runtime elements and their relationships. Runtime model defines one of the so-called KDM views that support separation of concerns in KDM models.

In the meta-model, RuntimeModel is a subclass of KDMModel. RuntimeModel is associated with the Root element.

Superclass

KDMModel

Associations

runtimeElement:RuntimeElement[0..*]

relation:RuntimeRelation[0..*]

Constraints

Semantics

20.4.2 RuntimeElement Class (abstract)

The RuntimeElement is a generic meta-model element that represents physical entities of the operating environments of software systems. In the meta-model a RuntimeElement is a subclass of KDMEntity.

Superclass

KDMEntity

Associations

runtimeGroup:RuntimeGroup[0..*]	The set of Runtime groups with which the current element is associated.

Constraints

Semantics

20.4.3 RuntimeContainer Class (abstract)

The RuntimeContainer is a meta-model element that provides generic grouping capabilities for RuntimeElements. RuntimeContainer owns a set of RuntimeElements. A RuntimeElement can be associated with a single RuntimeContainer.

In the meta-model RuntimeContainer is a subclass of KDMContainer. It is also a subclass of RuntimeElement. Several concrete KDM Runtime container classes further subclass RuntimeContainer class.

A RuntimeElement can be owned by at most one RuntimeContainer. The optional character of ownership is intended as a convenience to tools, allowing them to create RuntimeElements prior to linking them to the owning RuntimeContainer. RuntimeElements without an owner RuntimeContainer also capture the top-level element (especially RuntimeContainers and RuntimeGroups).

Superclass

KDMContainer
RuntimeElement

Constraints

Semantics

20.4.4 RuntimeGroup Class

The RuntimeGroup is a meta-model element that provides generic grouping capabilities for RuntimeElements. RuntimeGroup is associated with a set of RuntimeElements. A RuntimeElement can be associated with multiple RuntimeGroups. In the meta-model RuntimeGroup is a subclass of KDMGroup.

Superclass

KDMGroup
RuntimeElement

Associations

runtimeElement:RuntimeElement[0..*]	

Constraints

Semantics

20.5 Runnable Class Diagram

The Runnable class diagram defines meta-model elements that represent dynamic structures (instances of some logical entities and their relationships) that emerge at the so-called "run time" of the software system. For example, dynamic entities include processes and threads. Instances of processes and threads can be created dynamically and in many cases relations between the dynamically created instances of processes and threads are an essential part of the knowledge of existing systems. Another example of dynamic structures involves deployed components that are loaded dynamically.

The classes and associations that make up the Runnable diagram are shown in Figure 20.3.

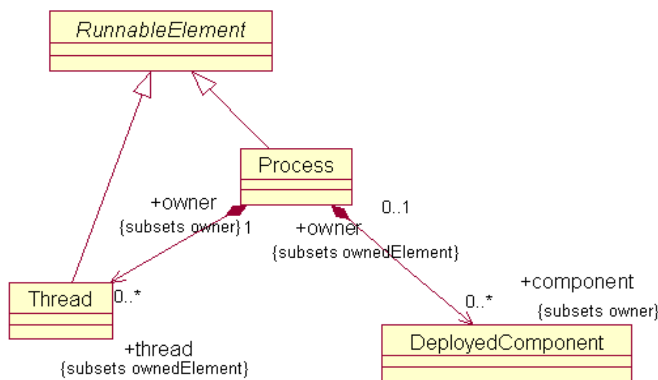

Figure 20.3 - Runnable Class Diagram

20.5.1 RunnableElement (abstract)

The RunnableElement is a generic meta-model element that represents an entity that has its own execution thread (for example, process or thread). RunnableElement is subclassed by Process and Thread. In the meta-model RunnableElement is used as the endpoint of certain RuntimeRelations.

Superclass

RuntimeContainer

Semantics

20.5.2 Process Class

The Process is a meta-model element that represents instances of processes.

In the meta-model Process is a subclass of RuntimeComponent. It is also a subclass of RunnableElement, which is a common superclass of both a Process and a Thread. In the meta-model RunnableElement is used as the endpoint of certain RuntimeRelations.

Superclass

RunnableElement

Associations

thread:Thread[0..*]

component:DeployedComponent[0..*]

Constraints

Semantics

20.5.3 Thread Class

The Thread is a meta-model element that represents instances of the so-called threads (light-weight processes).

In the meta-model Thread is a subclass of a RuntimeContainer. It is also a subclass of RunnableElement, which is a common superclass of both a Process and a Thread. In the meta-model RunnableElement is used as the endpoint of certain RuntimeRelations.

Superclass

RunnableElement

Constraints

Semantics

20.6 Deployment Class Diagram

The Deployment class diagram defines meta-model elements that represent deployment elements and their relations. In particular, the elements of the Deployment diagram address physical structures and how logical components are mapped to these physical structures.

The classes and associations that make up the Deployment diagram are shown in Figure 20.4.

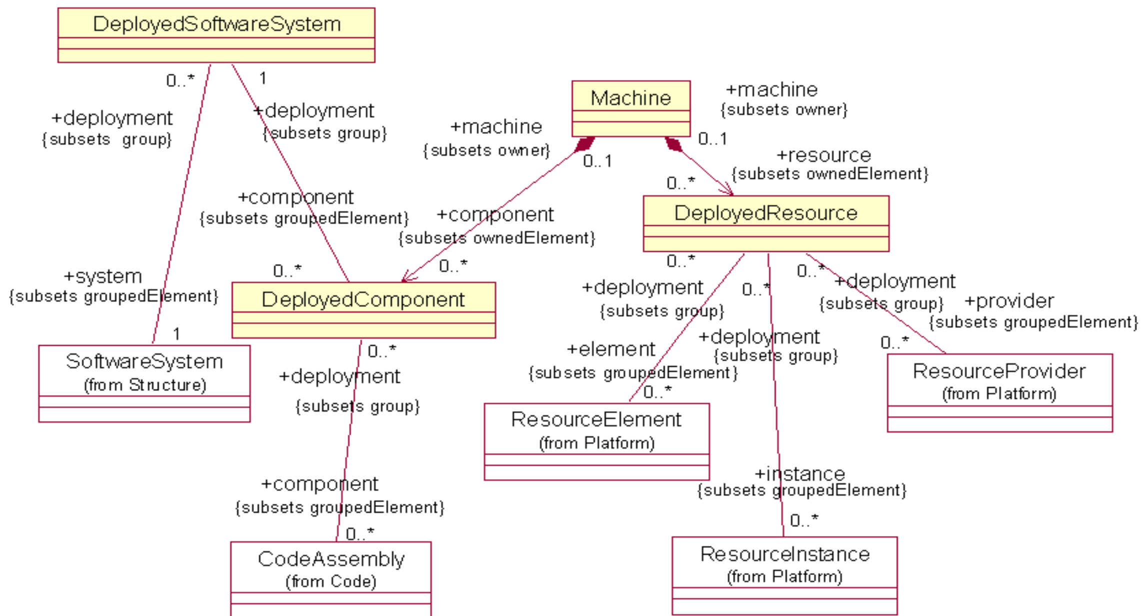

Figure 20.4 - Deployment Class Diagram

20.6.1 DeployedComponent Class

The DeployedComponent represents a unit of deployment as defined by a particular platform. Major platform parts provide a packaging mechanism of deploying application functionality. Deployment component is a replaceable unit of an application. For example, DLL, shared library, COM, Eclipse plugin, Executable, Jar file, War file for Tomcat, SQL Stored procedure, CORBA module, EJB, JavaBean, Jakarta Struts Action, Jakarta Struts Form.

In the meta-model DeployedComponent is a subclass of RuntimeGroup.

Superclass

RuntimeGroup

Associations

component:Code::CodeAssembly[0..*]	The code components which are deployed to the target DeployedComponent.
machine:Machine[0..1]	The Machine onto which the target DeployedComponent is deployed.
Deployment:DeployedSoftwareSystem [1]	

Constraints

Semantics

20.6.2 DeployedSoftwareSystem Class

The DeployedSoftwareSystem is a meta-model element that represents an instance of a software system at deployment or at the initialization time. DeployedSoftwareSystem is a physical instance of some logical SoftwareSystem. DeployedSoftwareSystem is associated with a set of DeployedComponents, which correspond to the set of logical components of the logical SoftwareSystem. The logical view of KDM model describes one or more SoftwareSystems. Each SoftwareSystem involves one or more Components. Some components can be involved in more than one SoftwareSystem (allowing description of the so-called Software Product Lines). Each Component involves one or more model Modules. Again, each Module can be involved in more than one Component. Component is a unit of deployment. Each logical component can be deployed multiple times, each time represented by a unique DeploymentComponent element. DeployedSoftwareSystem is a counterpart of the corresponding logical SoftwareSystem.

Superclass

RuntimeGroup

Associations

system:Structure::SoftwareSystem[0..1]	The logical system that is deployed.
component:DeployedComponent[0..*]	The set of physical DeployedComponents that make up the target system. The physical components correspond to the logical components of the system.

Constraints

Semantics

20.6.3 Machine Class

The Machine is a meta-model element that represents the hardware node which hosts deployed components. In the meta-model Machine is a subclass of RuntimeContainer.

Superclass

RuntimeContainer

Associations

component:DeployedComponent[0..*]

resource:DeployedResource[0..*]

Constraints

Semantics

20.6.4 DeployedResource Class

The DeployedResource is a meta-model element that represents a set of platform resource instances as they are deployed in a particular deployment configuration. DeployedResource is associated with a set of ResourceElements, ResourceInstances, and PlatformProviders. DeployedResource provides a unique physical context for a logical resource, as each logical resource can be associated with multiple DeployedResource.

In the meta-model Deployed Resource is a subclass of RuntimeGroup.

Superclass

RuntimeGroup

Associations

Machine:Machine[1]	The Machine onto which the target DeployedResource is deployed.
element:Platform::ResourceElement[0..*]	The set of ResourceElements which are deployed into the target DeployedResource.
instance:Platform::ResourceInstance[0..*]	The set of ResourceInstances which are deployed into the target DeployedResource.
provider:Platform::ResourceProvider[0..*]	The set of PlatformProviders which are deployed into the target DeployedResource.

Constraints

Semantics

20.7 RuntimeActions Class Diagram

The RuntimeActions class diagram defines meta-model elements that represent specific Runtime actions as the endpoints of certain Runtime relationships. The elements of this diagram extend ActionElement from the KDM Action package.

The classes and associations that make up the RuntimeActions diagram are shown in Figure 20.5.

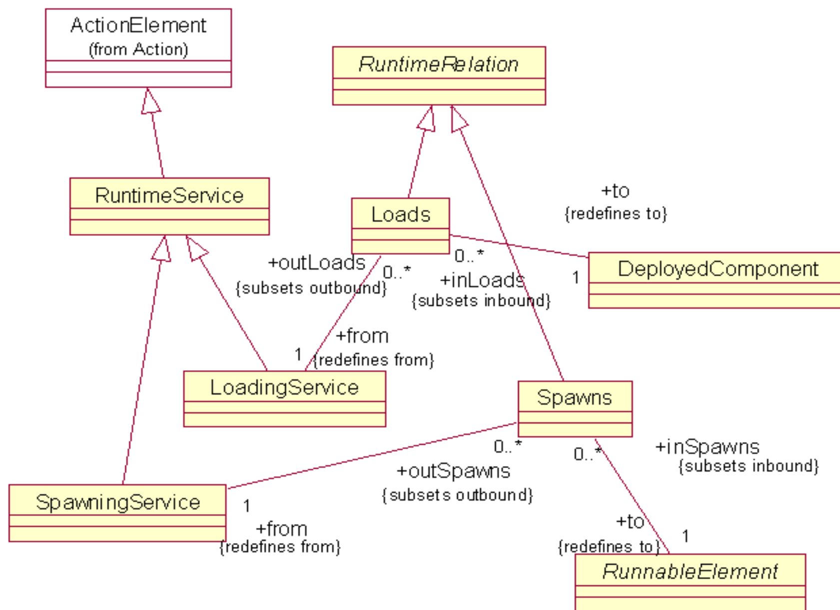

Figure 20.5 - RuntimeActions Class Diagram

20.7.1 RuntimeRelation Class (abstract)

The RuntimeRelation is a generic meta-model element that represents various relationships between the physical instances at Runtime.

In the meta-model the RuntimeRelations class is a subclass of KDMRelationship. This class is an extension point. In KDM RuntimeRelations is subclassed by several specific concrete relationships.

Superclass

KDMRelationship

Semantics

20.7.2 RuntimeService Class

The RuntimeService is a generic meta-model element that represents various endpoints of RuntimeRelation.

In the meta-model RuntimeService is a subclass of ActionElement. This class is an extension point. In KDM RuntimeService is subclassed by several specific concrete actions that correspond to specific RuntimeRelations.

Superclass

ActionElement

Semantics

20.7.3 LoadingService Class

The LoadingService is a meta-model element that represents the endpoint of Loads relation. In the meta-model LoadingService is a subclass of a generic element RuntimeService.

Superclass

RuntimeService

Associations

outLoads:Loads[0..*]	The set of outbound Loads relationships to DeployedComponents.

Constraints

Semantics

20.7.4 Loads Class

The Loads class is a meta-model element that represents "dynamic loading relationship" between a LoadingService action endpoint and the DeployedComponent.

In the meta-model Loads is a subclass of a generic element RuntimeRelation.

Superclass

RuntimeRelation

Associations

from:LoadingService[1]	The LoadingService endpoint
to:DeployedComponent[1]	The DeployedComponent which is being loaded.

Constraints

Semantics

20.7.5 SpawningService Class

The SpawningService is a meta-model element that represents the endpoint of Spawns relation. In the meta-model SpawningService is a subclass of generic element RuntimeService.

Superclass

RuntimeService

Associations

outSpawns:Spawns[0..*]	The set of outbound Spawns relationships to RunnableInterface.

Constraints

Semantics

20.7.6 Spawns Class

The Spawns class is a meta-model element that represents "dynamic process creation" or "dynamic thread creation" relationship between a SpawningService action endpoint and the RunnableInterface (Process or Thread). In the meta-model Spawns is a subclass of a generic element RuntimeRelationship.

Superclass

RuntimeRelation

Associations

from:SpawningService[1]	The SpawningService endpoint
to:RunnableElement[1]	The Runnable element (Process or Thread) which is being spawned.

Constraints

Semantics

20.7.7 DeployedComponent (additional properties)

Associations

inLoads:Loads[0..*]	The set of inbound Loads relationships to DeployedComponent.

Constraints

Semantics

20.7.8 RunnableElement (additional properties)

Associations

inSpawns:Spawns[0..*]	The set of inbound Spawns relationships to RunnableInterface.

Constraints

Semantics

21 Conceptual Package

21.1 Overview

The Conceptual Model defined in the KDM Conceptual package provides constructs for creating a conceptual model during the analysis phase of knowledge discovery from existing code.

The Conceptual Model enables mapping of KDM compliant model to models compliant to other specifications. Currently, it provides “concept” classes – TermUnit and FactUnit facilitating mapping to SBVR.

Future versions of KDM Conceptual package may provide additional “concept” classes facilitating mapping to other specifications. These meta-model objects and relationships between them will be used to define mapping between KDM “concepts” and SBVR.

The KDM Conceptual Model refers to two “concepts”: Term and Fact. The following is a mapping of this KDM “concepts” to the SBVR terminology:

- Term -> SBVR Noun (collectively referring to SBVR Terms and SBVR Names)
- Fact -> SBVR Fact

The SBVR vocabulary “concepts” (i.e., Term and Fact) are not defined in KDMInstead, the KDM Conceptual Model defines the implementations of these “concepts” - TermUnit and FactUnit. The mapping between KDM and SBVR is facilitated with the help of (0..*) to (0..*) relationships between pairs (i.e., <Term, TermUnit> and <Fact, FactUnit>) as shown in Figure 21.1.

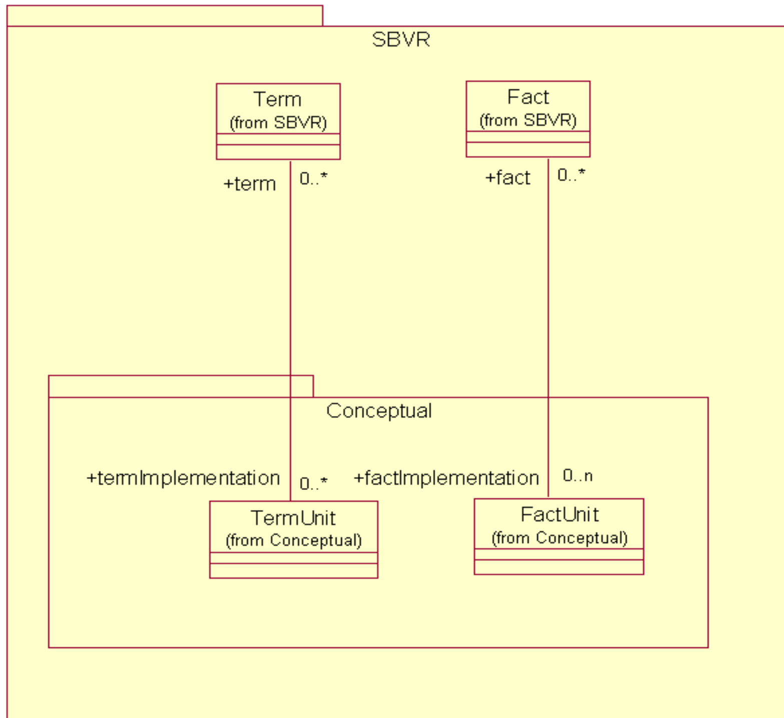

Figure 21.1 - Mapping between KDM and SBVR

21.2 Organization of the Conceptual Package

The Conceptual package is a collection of classes and associations that are described together because they provide meta-model constructs for defining. The Conceptual package depends on the following packages:

org.omg::ADM::KDM::Core

org.omg::ADM::KDM::Code

21.3 ConceptualInheritances Class Diagram

The ConceptualInheritances class diagram describes the inheritance relationships between children classes introduced in the Conceptual package and the parent classes defined in other packages. The ConceptualInheritances class diagram is shown in Figure 21.2.

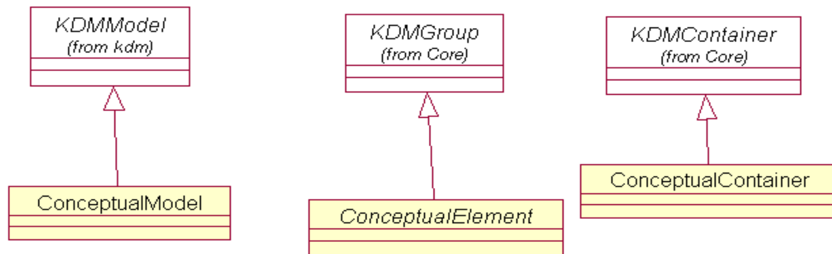

Figure 21.2 - ConceptualInheritances Class Diagram

21.4 ConceptualModel Class Diagram

The ConceptualModel class diagram collects together all classes and associations of the Conceptual package. They provide basic meta-model constructs to define specialized "concept" types. These meta-model objects and relationships between them will be used as a foundation for a conceptual model built by a mining tool as a result of knowledge discovery from existing code.

The class diagram shown in Figure 21.3 captures these classes and their relations.

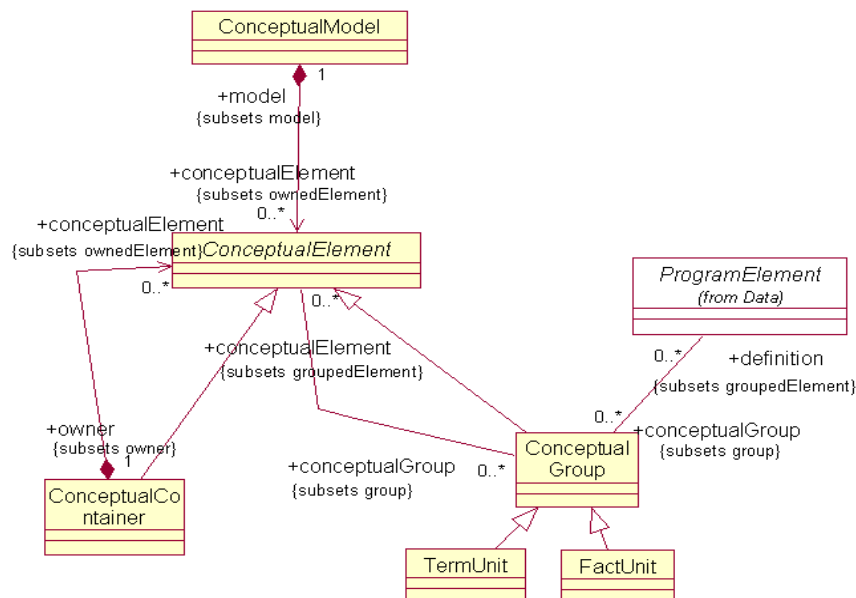

Figure 21.3 - ConceptualModel Class Diagram

21.4.1 ConceptualModel

The ConceptualModel holds the metadata of the conceptual model created by a mining tool.

Superclass

Core::KDMModel

Associations

conceptualElement:ConceptualElement[0..*]	Identifies the root "concept" elements of the hierarchy of the conceptual elements contained in the model. The ConceptualModel can contain zero or more such trees.

Constraints

Semantics

21.4.2 ConceptualElement (abstract)

The ConceptualElement class is used to facilitate a hierarchy and grouping of "concepts" within Conceptual Model.

Superclass

Core::KDMEntity

Associations

conceptualGroup:ConceptualElement[0..*]	Recursively identifies the embedded "concept" element. This containment relationship enables creation of a tree-like hierarchy of conceptual elements.

Constraints

Semantics

21.4.3 ConceptualContainer

The ConceptualContainer class is used to facilitate a hierarchy and grouping of "concepts" within Conceptual Model.

Superclass

Core::KDMEntity

Associations

conceptualElement:ConceptualElement[0..*]	Recursively identifies the embedded "concept" element. This containment relationship enables creation of a tree-like hierarchy of conceptual elements.

Constraints

Semantics

21.4.4 ConceptualGroup

The ConceptualGroup class is used to facilitate a hierarchy and grouping of “concepts” within Conceptual Model.

Superclass

Core::KDMEntity

Associations

conceptualGroup:ConceptualElement[0..*]	Recursively identifies the embedded “concept” element. This containment relationship enables creation of a tree-like hierarchy of conceptual elements.
definition:ProgramElement[0..*]	

Constraints

Semantics

21.4.5 TermUnit

The TermUnit class represents an implementation of a “concept” used for mapping to the SBVR Term.

Superclass

ConceptualGroup

Semantics

21.4.6 FactUnit

The FactUnit class represents an implementation of a “concept” used for mapping to the SBVR Fact.

Superclass

ConceptualGroup

Semantics

22 Behavior Package

22.1 Overview

The Behavior Model defined in the KDM Behavior package provides constructs for creating a behavior model during the analysis phase of knowledge discovery from existing code.

The BehaviorModel enables mapping of KDM compliant model to models compliant to other specifications. It provides "behavior" types – BehaviorUnit, ScenarioUnit and RuleUnit facilitating mapping to various external models including but not limited to activities/flow chart and swim lane diagrams, use case scenarios, SBVR, etc.

The following explains the difference between these "behavior" types:

> **BehaviorUnit** represents a behavior graph with all possible paths through the application logic and associated conditions. The conditions responsible for navigation between alternative paths within the graph are supported by RuleUnits.
>
> **ScenarioUnit** represents a singe path through the behavior graph where all navigational conditions are resolved.
>
> **RuleUnit** represents a condition, group of conditions, or constraint.
>
> **RuleUnit** responsible for navigation within behavior graph controls the BehaviorUnit. If these conditions are resolved we are dealing with a ScenarioUnit.
>
> **RuleUnit** type supports mapping to SBVR. The KDM RuleUnit is mapped to the SBVR Rule.

The SBVR Rule is not defined in KDM. Instead, the KDM BehaviorModel package defines the implementations of the SBVR Rule - RuleUnit. The mapping between KDM RuleUnit and an SBVR Rule is facilitated with the help of (0..*) to (0..*) relationships between them as shown in Figure 22.1.

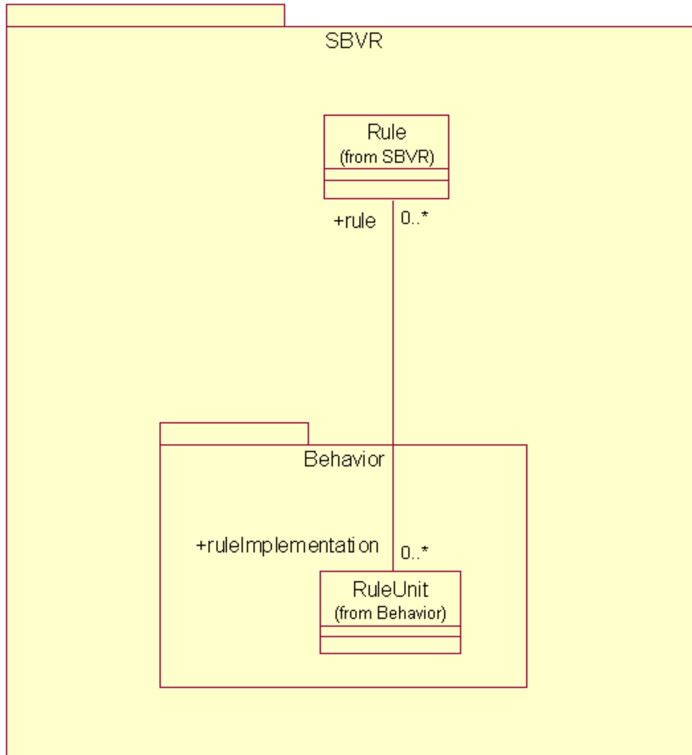

Figure 22.1 - Mapping between KDM and SBVR

22.2 Organization of the Behavior Package

The Behavior package is a collection of classes and associations that are described together because they provide meta-model constructs for defining. The Behavior package depends on the following packages:

org.omg::ADM::KDM::Core

org.omg::ADM::KDM::Action

org.omg::ADM::KDM::Code

22.3 BehaviorInheritances Class Diagram

The BehaviorInheritances class diagram describes the inheritance relationships between children classes introduced in the Behavior package and the parent classes defined in other packages. The BehaviorInheritances class diagram is shown in Figure 22.2.

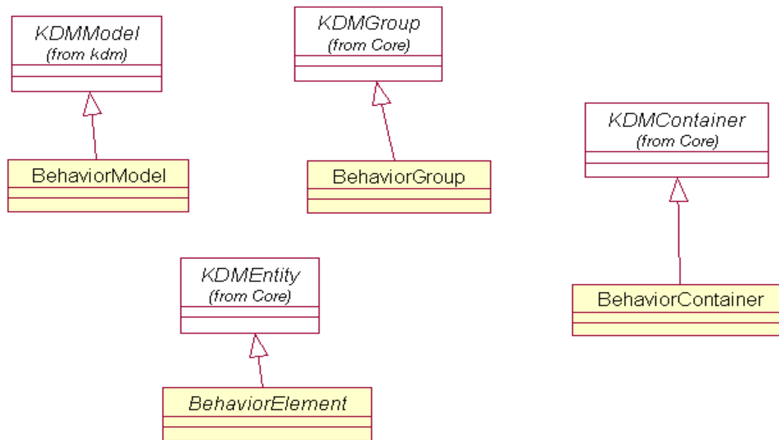

Figure 22.2 - BehaviorInheritances Class Diagram

22.4 BehaviorModel Class Diagram

The BehaviorModel class diagram collects together all classes and associations of the Behavior package. They provide basic meta-model constructs to define specialized "behavior" types. These meta-model objects and relationships between them will be used as a foundation for a behavior model built by a mining tool as a result of knowledge discovery from existing code.

The class diagram shown in Figure 22.3 captures these classes and their relations.

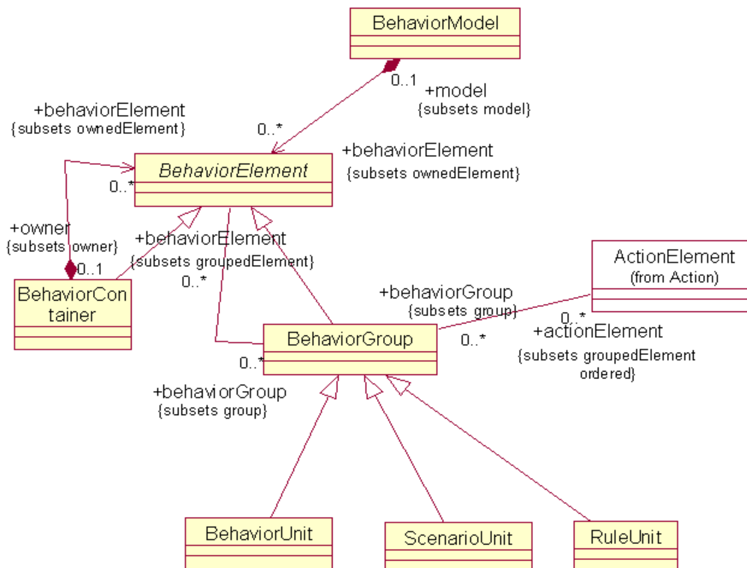

Figure 22.3 - BehaviorModel Class Diagram

22.4.1 BehaviorModel

The BehaviorModel holds the metadata of the behavior model created by a mining tool.

Superclass

Core::KDMModel

Associations

behaviorElement:BehaviorElement[0..*]	Identifies the root behavior element of the hierarchy of the behavior entities making contained in the model. The Behavior Model can contain zero or more such trees.

Constraints

Semantics

22.4.2 BehaviorElement (abstract)

The BehaviorElement class is used to facilitate a hierarchy and grouping of "behavior" types within Behavior package.

Superclass

Core::KDMEntity

Associations

behaviorGroup:BehaviorGroup[0..*]	Recursively identifies the embedded behavior element. This containment relationship enables creation of a tree-like hierarchy of behavior elements.

Constraints

Semantics

22.4.3 BehaviorGroup

The BehaviorGroup class is used to facilitate a hierarchy and grouping of "behavior" types within Behavior package.

Superclass

Core::KDMEntity

Associations

behaviorElement:BehaviorElement[0..*]	Recursively identifies the embedded behavior element. This containment relationship enables creation of a tree-like hierarchy of behavior elements.

actions:Action::ActionElement[0..*]{ordered}	Identifies the action element. This containment relationship provides traceability from the behavior elements to the actions, which make up these behavior entities. Action elements in turn are associated with the code (Code::CodeElement) which actually implements them. That is defined in the Action package.

Constraints

Semantics

22.4.4 BehaviorContainer

The BehaviorContainer class is used to facilitate a hierarchy and grouping of "behavior" types within Behavior package.

Superclass

Core::KDMContainer

Associations

behaviorElement:BehaviorElement[0..*]	Recursively identifies the embedded behavior element. This containment relationship enables creation of a tree-like hierarchy of behavior elements.

Constraints

Semantics

22.4.5 BehaviorUnit

The BehaviorUnit class represents a description of the behavior graph with multiple paths and associated conditions.

Superclass

BehaviorGroup

Semantics

22.4.6 ScenarioUnit

The ScenarioUnit class represents a description of a single path through a behavior graph where all graph navigation conditions are resolved.

Superclass

BehaviorGroup

Semantics

22.4.7 RuleUnit

The RuleUnit class represents a description of a condition or a constraint implemented in the existing code and discovered by a Mining Tool.

Superclass

BehaviorGroup

Semantics