UML Semantics

version 1.1
1 September 1997

Rational Software ■ Microsoft ■ Hewlett-Packard ■ Oracle
Sterling Software ■ MCI Systemhouse ■ Unisys ■ ICON Computing
IntelliCorp ■ i-Logix ■ IBM ■ ObjecTime ■ Platinum Technology ■ Ptech
Taskon ■ Reich Technologies ■ Softeam

ad/97-08-04

Copyright © 1997 Rational Software Corporation.
Copyright © 1997 Microsoft Corporation.
Copyright © 1997 Hewlett-Packard Company.
Copyright © 1997 Oracle Corporation.
Copyright © 1997 Sterling Software.
Copyright © 1997 MCI Systemhouse Corporation.
Copyright © 1997 Unisys Corporation.
Copyright © 1997 ICON Computing.
Copyright © 1997 IntelliCorp.
Copyright © 1997 i-Logix.
Copyright © 1997 IBM Corporation.
Copyright © 1997 ObjecTime Limited.
Copyright © 1997 Platinum Technology, Inc.
Copyright © 1997 Ptech Inc.
Copyright © 1997 Taskon A/S.
Copyright © 1997 Reich Technologies.
Copyright © 1997 Softeam.

Photocopying, electronic distribution, or foreign-language translation of this document is permitted, provided this document is reproduced in its entirety and accompanied with this entire notice, including the following statement:

> The most recent updates on the Unified Modeling Language are available via the worldwide web, www.rational.com/uml.

The UML logo is a trademark of Rational Software Corp.
OMG, CORBA, CORBAfacility, and IDL are trademarks of the Object Management Group, Inc.

Contents

1. INTRODUCTION 1

1.1 Purpose 1
1.2 Scope 1
1.3 Approach 2
1.4 Document Organization 3
1.5 Related Documents 3

PART 1. BACKGROUND 5

2. LANGUAGE ARCHITECTURE 6

2.1 Overview 6
2.2 Four-Layer Metamodel Architecture 6
2.3 Package Structure 8

3. LANGUAGE FORMALISM 10

3.1 Levels of Formalism 10
3.2 Package Specification Structure 11
3.3 Use of a Constraint Language 12
3.4 Use of Natural Language 12
3.5 Naming Conventions and Typography 13

PART 2: FOUNDATION 14

4. FOUNDATION PACKAGE: CORE 15

4.1 Overview 15
4.2 Abstract Syntax 15
4.3 Well-Formedness Rules 27
4.4 Semantics 34
4.5 Standard Elements 39
4.6 Notes 40

5. FOUNDATION PACKAGE: AUXILIARY ELEMENTS 42

5.1 Overview 42
5.2 Abstract Syntax 43
5.3 Well-Formedness Rules 47
5.4 Semantics 49
5.5 Standard Elements 49

6. FOUNDATION PACKAGE: EXTENSION MECHANISMS 51

6.1 Overview 51
6.2 Abstract Syntax 53
6.3 Well-Formedness Rules 56
6.4 Semantics 57
6.5 Standard Elements 58
6.6 Notes 58

7. FOUNDATION PACKAGE: DATA TYPES 59

7.1 Overview 59
7.2 Abstract Syntax 60

PART 3. BEHAVIORAL ELEMENTS 64

8. BEHAVIORAL ELEMENTS PACKAGE: COMMON BEHAVIOR 65

8.1 Overview 65
8.2 Abstract Syntax 66
8.3 Well-Formedness Rules 73
8.4 Semantics 77
8.5 Standard Elements 79

9. BEHAVIORAL ELEMENTS PACKAGE: COLLABORATIONS 80

9.1 Overview 80
9.2 Abstract Syntax 81
9.3 Well-Formedness Rules 84
9.4 Semantics 86
9.4 Standard Elements 88
9.5 Notes 88

10. BEHAVIORAL ELEMENTS PACKAGE: USE CASES 89

10.1 Overview 89
10.2 Abstract Syntax 90
10.3 Well-Formedness Rules 91
10.4 Semantics 92
10.5 Standard Elements 96
10.6 Notes 96

11. BEHAVIORAL ELEMENTS PACKAGE: STATE MACHINES 97

11.1 Overview 97
11.2 Abstract Syntax 97
11.3 Well-Formedness Rules 103
11.4 Semantics 106
11.5 Standard Elements 114
11.6 Notes 115
11.7 Activity Models 120

PART 4. GENERAL MECHANISMS 127

12. MODEL MANAGEMENT PACKAGE 128

12.1 Overview 128
12.2 Abstract Syntax 129
12.3 Well-Formedness Rules 131
12.4 Semantics 134
12.5 Standard Elements 137
12.5 Notes 137

APPENDICES 138

APPENDIX A: STANDARD ELEMENTS 139

A.1 Stereotypes 139
A.2 Tagged Values 144
A.3 Constraints 145

APPENDIX B: GLOSSARY 147

INDEX 161

Figures

Figure 1: Top-Level Packages *8*
Figure 2: Foundation Packages *9*
Figure 3: Behavioral Elements Packages *9*
Figure 4: Foundation Packages *14*
Figure 5: Core Package–Backbone *16*
Figure 6: Core Package–Relationships *17*
Figure 7: Auxiliary Elements—Dependencies and Templates *43*
Figure 8: Auxiliary Elements—Physical Structures and View Elements *44*
Figure 9: Extension Mechanisms *53*
Figure 10: Data Types *60*
Figure 11: Behavioral Elements Packages *64*
Figure 12: Common Behavior—Requests *66*
Figure 13: Common Behavior—Actions *67*
Figure 14: Common Behavior—Instances and Links *67*
Figure 15: Collaborations *81*
Figure 16: Use Cases *90*
Figure 17: State Machines—Main *98*
Figure 18: State Machines—Events *98*
Figure 19: Modeling Class Behavior Example *115*
Figure 20: State Machine for Modeling Class Behavior *116*
Figure 21: State Machine Refinement Example *116*
Figure 22: Activity Models *121*
Figure 23: Model Management *129*

Tables

Table 1: Four Layer Metamodeling Architecture *7*
Table 2: Core—Standard Elements *40*
Table 3: Auxiliary Elements—Standard Elements *50*
Table 4: Common Behavior—Standard Elements *79*
Table 5: State Machines—Standard Elements *114*
Table 6: Model Management—Standard Elements *137*

1. INTRODUCTION

The Unified Modeling Language (UML) is a general-purpose visual modeling language that is designed to specify, visualize, construct and document the artifacts of a software system. The UML is simple and powerful. The language is based on a small number of core concepts that most object-oriented developers can easily learn and apply. The core concepts can be combined and extended so that expert object modelers can define large and complex systems across a wide range of domains.

The UML specification consists of two interrelated parts:

- *UML Semantics*. A metamodel that specifies the abstract syntax and semantics of UML object modeling concepts.

- *UML Notation*. A graphic notation for the visual representation of the UML semantics.

The abstract syntax for the UML Semantics is expressed using a small subset of the UML Notation. In addition, the UML Notation describes the mapping of the graphic notation to the underlying semantics. In this manner the two parts complement each other without duplicating functionality. Whenever there is a difference between the two parts regarding semantics, this document takes precedence.

This document specifies the UML Semantics. It describes the semantics using a metamodel that is described in three views: abstract syntax, well-formedness rules and semantics.

Feedback regarding this document should be e-mailed to uml_feedback@rational.com.

1.1 PURPOSE

This document is primarily intended as a comprehensive and precise specification of the UML's semantic constructs. The primary audience for this detailed description consists of the OMG, other standards organizations, tool builders, metamodelers, methodologists, and expert modelers. The authors assume familiarity with metamodeling and advanced object modeling. Readers looking for an introduction to the UML or object modeling should consider another source.

Although the document is meant for advanced readers, it is also meant to be easily understood by its intended audience. Consequently it is structured and written to increase readability. The structure of the document, like the language, builds on previous concepts to refine and extend the semantics. In addition, the document is written in a 'semi-formal' style that combines natural and formal languages in a complementary manner.

1.2 SCOPE

This document specifies semantics for structural and behavioral object models. Structural models (also known as static models) emphasize the structure of objects in a system, including their classes, interfaces, attributes and relations. Behavioral models (also known as dynamic models) emphasize the behavior of objects in a system, including their methods, interactions, collaborations and state histories.

The specification provides complete semantics for all modeling notations described in the *UML Notation* document. This includes support for a wide range of diagram techniques: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, and deployment diagram. The *UML Notation* document includes a summary of the semantics sections that are relevant to each diagram technique.

1.3 APPROACH

The specification emphasizes language architecture and formal rigor. The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers: user objects, model, metamodel, and meta-metamodel. This document is primarily concerned with the metamodel layer, which is an instance of the meta-metamodel layer. For example, *Class* in the metamodel is an instance of *MetaClass* in the meta-metamodel. The metamodel architecture of UML is discussed further in the *Language Architecture* section.

The UML metamodel is a logical model and not a physical (or implementation) model. The advantage of a logical metamodel is that it emphasizes declarative semantics, and suppresses implementation details. Implementations that use the logical metamodel must conform to its semantics, and must be able to import and export full as well as partial models. However, tool vendors may construct the logical metamodel in various ways, so they can tune their implementations for reliability and performance. The disadvantage of a logical model is that it lacks the imperative semantics required for accurate and efficient implementation. Consequently, the metamodel is accompanied with implementation notes for tool builders.

UML is also structured within the metamodel layer. The language is decomposed into several logical packages: Foundation, Behavioral Elements, and General Mechanisms. These packages in turn are decomposed into subpackages. For example, the Foundation package consists of the Core, Auxiliary Elements, Extension Mechanisms and Data Types subpackages. The structure of the language is fully described in the *Language Architecture* section.

The metamodel is described in a semi-formal manner using three views:

- Abstract syntax
- Well-formedness rules
- Semantics

The abstract syntax is provided as a model described in a subset of UML, consisting of a UML class diagram and a supporting natural language description. (In this way the UML bootstraps itself in a manner similar to how a compiler is used to compile itself.) The well-formedness rules are provided using a formal language (Object Constraint Language) and natural language (English). Finally, the semantics are described primarily in natural language, but may include some additional notation, depending on the part of the model being described. The adaptation of formal techniques to specify the language is fully described in the *Language Formalism* section

In summary, the UML metamodel is described in a combination of graphic notation, natural language and formal language. We recognize that there are theoretical limits to what one can express about a metamodel using the metamodel itself. However, our experience suggests that this combination strikes a reasonable balance between expressiveness and readability.

1.4 DOCUMENT ORGANIZATION

This document consists of several parts:

- Part 1: Background, sections 2 – 3.
- Part 2: Foundation, sections 4 – 7.
- Part 3: Behavioral Elements, sections 8 – 11.
- Part 4: General Mechanisms, section 12.

Part 1 explains how the UML is structured and specified. The language architecture section describes the structure of the language and explains its four-layer metamodel architecture. The language specification section describes how the language is rigorously defined using multiple views.

Part 2 defines the infrastructure for UML, the Foundation package. The Foundation package is decomposed into several subpackages: Core, Auxiliary Elements, Extension Mechanisms and Data Types. The Core package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as metaclasses, metaassociations, and metaattributes. The Auxiliary Elements package defines additional constructs that extend the Core to support advanced concepts such as dependencies, templates, physical structures and view elements. The Extension Mechanisms package specifies how model elements are customized and extended with new semantics. The Data Types package defines basic data structures for the language.

Part 3 defines the superstructure for behavioral modeling in UML, the Behavioral Elements package. The Behavioral Elements package consists of four lower-level packages: Common Behavior, Collaborations, Use Cases, and State Machines. Common Behavior specifies the core concepts required for behavioral elements. The Collaborations package specifies a behavioral context for using model elements to accomplish a particular task. The Use Case package specifies behavior using actors and use cases. The State Machines package defines behavior using finite-state transition systems.

Part 4 defines mechanisms of general applicability to models. This version of UML contains one general mechanisms package, Model Management. The Model Management package specifies how model elements are organized into models, packages and systems.

1.5 RELATED DOCUMENTS

The following documents are also important for understanding the UML metamodel and how it is used:

- The *UML Summary* provides an introduction to the UML, discussing its motivation and history.

- The *UML Notation Guide* defines the graphic syntax for expressing the semantics described by the UML metamodel. Consequently, the UML Notation Guide should be read in conjunction with the *UML Semantics* document.

- The *Object Constraint Language Specification* describes the OCL syntax, semantics, and grammar. All OCL features are described in terms of concepts from the UML semantics document.

- The *UML CORBAfacility Interface Definition* specifies a tool interoperability interface using CORBA IDL.

- The *UML Proposal Summary* summarizes the OMG proposal and discusses the relationship of UML to other technologies, including the MOF meta-metamodel.

PART 1. BACKGROUND

Part 1 explains how the UML is structured and specified. The language architecture section describes the structure of the language and explains its four-layer metamodel architecture. The language formalism section describes how the language is defined using three complementary views.

Contents

2. Language Architecture

3. Language Formalism

2. LANGUAGE ARCHITECTURE

2.1 OVERVIEW

The UML metamodel defines the complete semantics for representing object models using UML. It is defined in a metacircular manner, using a subset of UML notation and semantics to specify itself. In this way the UML metamodel bootstraps itself in a manner similar to how a compiler is used to compile itself.

The UML metamodel is one of the layers of a four-layer metamodel architecture. Since the metamodel layer is relatively complex it is decomposed into logical packages. The UML packages increase the modularity of the language and support multiple compliance points. The following sections provide an overview of the UML four-layer metamodel architecture and describe its package structure.

2.2 FOUR-LAYER METAMODEL ARCHITECTURE

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture. This architecture is a proven infrastructure for defining the precise semantics required by complex models. There are several other advantages associated with this approach:

- It validates core constructs by recursively applying them to successive metalayers.
- It provides an architectural basis for defining future UML metamodel extensions.
- It furnishes an architectural basis for aligning the UML metamodel with other standards based on a four-layer metamodeling architecture (e.g., the OMG Meta-Object Facility, CDIF).

The generally accepted conceptual framework for metamodeling is based on an architecture with four layers:

- meta-metamodel
- metamodel
- model
- user objects

These functions of these layers are summarized in the following table:

Table 1: Four Layer Metamodeling Architecture

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_Software_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel. A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes. A meta-metamodel can define multiple metamodels, and there can be multiple meta-metamodels associated with each metamodel.[1] While it is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs, this is not a strict rule. Each layer needs to maintain its own design integrity. Examples of meta-metaobjects in the meta-metamodeling layer are: *MetaClass*, *MetaAttribute*, and *MetaOperation*.

A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. Examples of metaobjects in the metamodeling layer are: *Class, Attribute, Operation,* and *Component.*

A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: *StockShare, askPrice, sellLimitOrder,* and *StockQuoteServer.*

User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain. Examples of objects in the user objects

[1] If there is not an explicit meta-metamodel, there is an implicit meta-metamodel associated with every metamodel.

layer are: *<Acme_Software_Share_98789>*, *654.56*, *sell_limit_order*, and *<Stock_Quote_Svr_32123>*.

The UML metamodel has been architected so that is can be instantiated from the OMG Meta Object Facility (MOF) meta-metamodel. The relationship of the UML metamodel to the MOF meta-metamodel is described in an appendix of the *UML Proposal Summary*.

2.3 PACKAGE STRUCTURE

The UML metamodel is moderately complex. It is composed of approximately 90 metaclasses and over 100 metaassociations, and includes almost 50 stereotypes. The complexity of the metamodel is managed by organizing it into logical packages. These packages group metaclasses that show strong cohesion with each other and loose coupling with metaclasses in other packages. The UML metamodel is decomposed into the top-level packages shown in Figure 1:

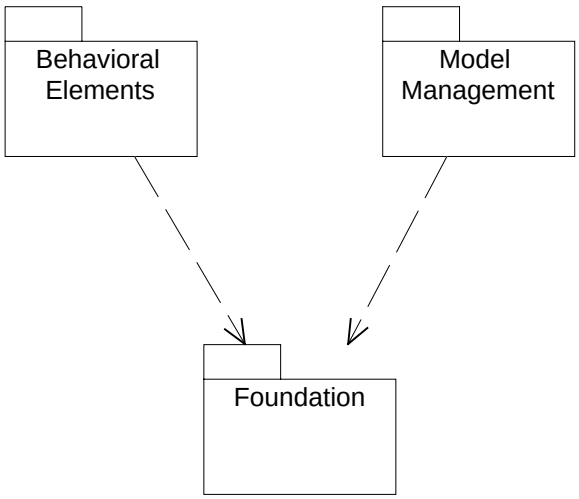


Figure 1: Top-Level Packages

The Foundation and Behaviorial Elements packages are further decomposed as shown in Figure 2 and Figure 3:

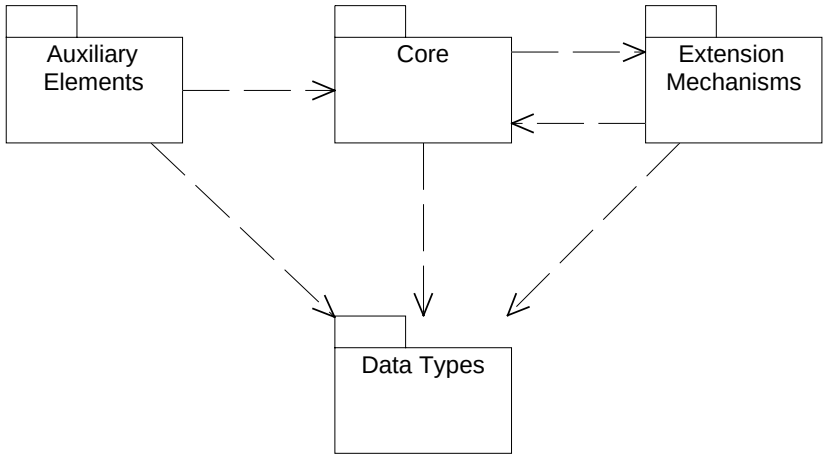


Figure 2: Foundation Packages

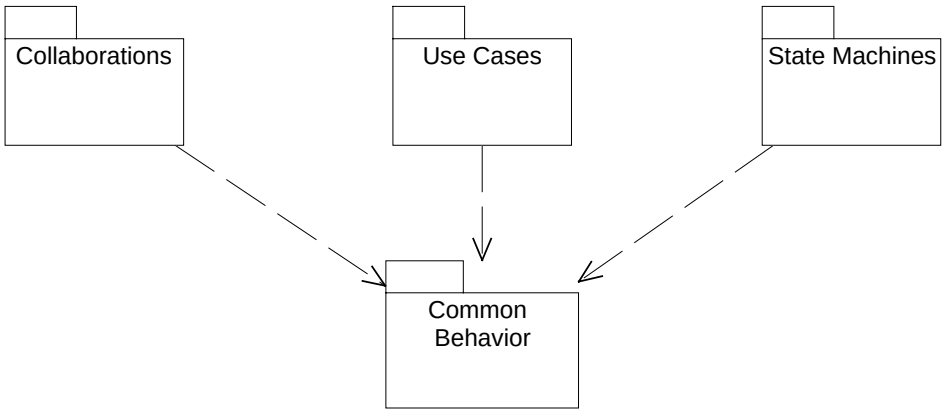


Figure 3: Behavioral Elements Packages

The functions and contents of these packages are described in Parts 2 – 4 of this document.

3. LANGUAGE FORMALISM

This section contains a description of the techniques used to describe UML. The specification adapts formal techniques to improve precision while maintaining readability. The technique describes the UML metamodel in three views using both text and graphic presentations. The benefits of adapting formal techniques include:

- the correctness of the description is improved,
- ambiguities and inconsistencies are reduced,
- the architecture of the metamodel is validated by a complementary technique, and
- the readability of the description is increased.

It is important to note that the current description is not a completely formal specification of the language because to do so would have added significant complexity without clear benefit. In addition, the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces.

The structure of the language is nevertheless given a precise specification, which is required for tool interoperability. The dynamic semantics are described using natural language, although in a precise way so they can easily be understood. Currently, the dynamic semantics are not considered essential for the development of tools. However, this will probably change in the future.

3.1 LEVELS OF FORMALISM

A common technique for specification of languages is to first define the syntax of the language and then to describe its static and dynamic semantics. The syntax defines what constructs exist in the language and how the constructs are built up in terms of other constructs. Sometimes, especially if the language has a graphic syntax, it is important to define the syntax in a notation independent way (i.e., to define the abstract syntax of the language). The concrete syntax is then defined by mapping the notation onto the abstract syntax. The syntax is described in sections headed *Abstract Syntax*.

The static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful, and the dynamic semantics define the meaning of a well-formed construct. The meaning of a description written in the language is defined only if the description is well formed (i.e., if it fulfills the rules defined in the static semantics). The static semantics are found in sections headed *Well-Formedness Rules*. The dynamic semantics are described under the heading *Semantics*. In some cases parts of the static semantics are also explained in the *Semantics* section for completeness.

The specification uses a combination of languages—a subset of UML, an object constraint language, and precise natural language—to describe the abstract syntax and semantics of the full UML. The description is self-contained; no other sources of information are needed to read the

document.[2] Although this is a metacircular description[3], understanding this document is practical since only a small subset of UML constructs are needed to describe its semantics.

In constructing the UML metamodel different techniques have been used to specify language constructs, using some of the capabilities of UML. The main language constructs are reified into metaclasses in the metamodel. Other constructs, in essence being variants of other ones, are defined as stereotypes of metaclasses in the metamodel. This mechanism allows the semantics of the variant construct to be significantly different from the base metaclass. Another more "lightweight" way of defining variants is to use metaattributes. As an example, the *aggregation* construct is specified by an attribute of the metaclass ***AssociationEnd***, which is used to indicate if an association is an ordinary aggregate, a composite aggregate, or a common association.

3.2 PACKAGE SPECIFICATION STRUCTURE

The document has one section for each package in the UML metamodel. Each of these sections has the following subsections:

- **Abstract Syntax** The abstract syntax is presented in a diagram showing the metaclasses defining the constructs and their relationships. The diagram also presents some of the well-formedness rules, mainly the multiplicity requirements of the relationships, and whether or not the instances of a particular sub-construct must be ordered. Finally, a short informal description in natural language describing each construct is supplied. The first paragraph of each of these descriptions is a general presentation of the construct which sets the context, while the following paragraphs give the informal definition of the metaclass specifying the construct in UML. For each metaclass, its attributes are enumerated together with a short explanation. Furthermore, the opposite role names of associations connected to the metaclass are also listed in the same way.

- **Well-Formedness Rules** The static semantics of each construct in UML, except for multiplicity and ordering constraints, are defined as a set of invariants of an instance of the metaclass. These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an OCL expression together with an informal explanation of the expression. In many cases, additional operations on the metaclasses are needed for the OCL expressions. These are then defined in a separate subsection after the well-formedness rules for the construct, using the same approach as the abstract syntax: an informal explanation followed by the OCL expression defining the operation.

  The statement 'No extra well-formedness rules' means that all current static semantics are expressed in the superclasses together with the multiplicity and type information expressed in the diagrams.

---

[2] Although a comprehension of the UML's four-layer metamodel architecture and its underlying meta-metamodel is helpful, it is not essential to understand the UML semantics.

[3] In order to understand the description of the UML semantics, you must understand some UML semantics.

- **Semantics** The meanings of the constructs are defined using natural language. The constructs are grouped into logical chunks that are defined together. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.

- **Standard Elements** Stereotypes of the metaclasses defined previously in the section are listed, with an informal definition in natural language. Well-formedness rules, if any, for the stereotypes are also defined in the same manner as in the *Well-Formedness Rules* subsection. Other kinds of standard elements (constraints and tagged-values) are listed, and are defined in the Standard Elements appendix.

- **Notes** This subsection may contain rationales for metamodeling decisions, pragmatics for the use of the constructs, and examples, all written in natural language.

A complete index appears at the end of the document. Index entries have the page number bold when the term is "first defined" (i.e. in the list of terms in an Abstract Syntax subsection). Later uses or extra definitions are in ordinary font.

3.3 USE OF A CONSTRAINT LANGUAGE

The specification uses the Object Constraint Language (OCL), as defined in *Object Constraint Language Specification*, for expressing well-formedness rules. The following conventions are used to promote readability:

- 'self', which can be omitted as a reference to the metaclass defining the context of the invariant, has been kept for clarity.

- In expressions where a collection is iterated, an iterator is used for clarity, even when formally unnecessary. The type of the iterator is usually omitted, but included when it adds to understanding.

- The 'collect' operation is left implicit where this is practical.

3.4 USE OF NATURAL LANGUAGE

We have striven to be precise in our use of natural language, in this case English. For example, the description of UML semantics includes phrases such as "X provides the ability to…" and "X is a Y." In each of these cases, the usual English meaning is assumed, although a deeply formal description would demand a specification of the semantics of even these simple phrases.

The following general rules apply:

- When referring to an instance of some metaclass, we often omit the word "instance". For example, instead of saying "a Class instance" or "an Association instance", we just say "a Class" or "an Association". By prefixing it with an "a" or "an", assume that we mean "an

instance of". In the same way, by saying something like "Elements" we mean "a set (or the set) of instances of the metaclass Element".

- Every time a word coinciding with the name of some construct in UML is used, that construct is referred.

- Terms including one of the prefixes sub, super, or meta are written as one word (e.g., metamodel, subclass).

3.5 NAMING CONVENTIONS AND TYPOGRAPHY

In the description of UML the following conventions have been used:

- When referring to constructs in UML, not their representation in the metamodel, normal text is used.

- The first time a UML construct or mechanism is mentioned in a context it is italicized.

- Metaclass names always start with uppercase, remainder lowercase, using italics and bold, e.g.: 'Classifier'. For names that consist of appended nouns/adjectives, initial embedded capitals are used (e.g. 'ModelElement', 'StructuralFeature').

- Names of metaassociations/association classes are written in the same manner as metaclasses, e.g. 'ElementReference'

- Lowercase italics are used for names of association ends (role names), attributes and operations in the metamodel, e.g.: 'type', 'feature'. Initial embedded capital is used for names that consist of appended nouns/adjectives e.g. 'ownedElement', 'allContents'.

- Boolean metaattribute names always start with 'is' (e.g. 'isAbstract').

- While referring to metaclasses, metaassociations, metaattributes, etc in the text, the exact names as they appear in the model are always used.

- Names of abstract metaclasses are shown in italics in the metamodel, but not distinguished from other metaclasses in the text.

- Names of stereotypes are delimited by guillemets and begin with lowercase (e.g., «type»).

PART 2: FOUNDATION

Part 2 defines the infrastructure for UML, the Foundation package. The Foundation package is decomposed into several subpackages: Core, Auxiliary Elements, Extension Mechanisms and Data Types. The Core package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as metaclasses, metaassociations, and metaattributes. The Auxiliary Elements package defines additional constructs that extend the Core to support advanced concepts such as dependencies, templates, physical structures and view elements. The Extension Mechanisms package specifies how model elements are customized and extended with new semantics. The Data Types package defines basic data structures for the language.

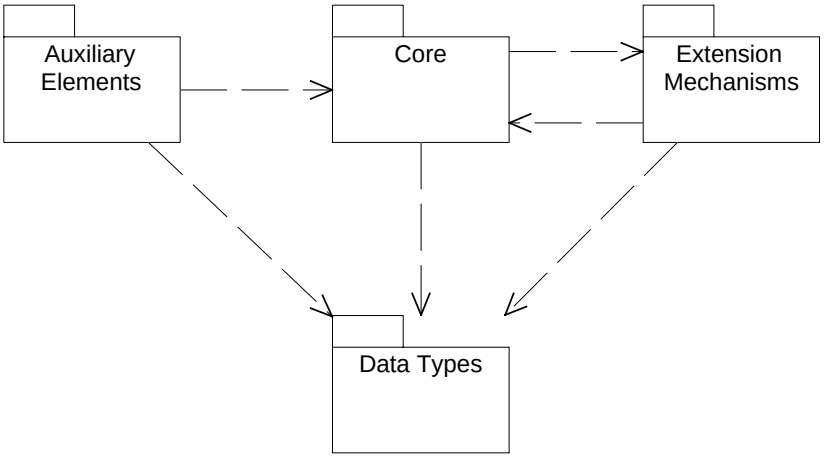


Figure 4: Foundation Packages

Contents

4. Foundation Package: Core

5. Foundation Package: Auxiliary Elements

6. Foundation Package: Extension Mechanisms

7. Foundation Package: Data types

4. FOUNDATION PACKAGE: CORE

4.1 OVERVIEW

The *Core* package is the most fundamental of the subpackages that compose the UML Foundation package. It defines the basic abstract and concrete constructs needed for the development of object models. Abstract metamodel constructs are not instantiable and are commonly used to reify key constructs, share structure, and organize the model. Concrete metamodel constructs are instantiable and reflect the modeling constructs used by object modelers (cf. metamodelers). Abstract constructs defined in the Core include ***ModelElement***, ***GeneralizableElement***, and ***Classifier***. Concrete constructs specified in the Core include ***Class***, ***Attribute***, ***Operation***, and ***Association***.

The Core package specifies the core constructs required for a basic metamodel and defines an architectural backbone ("skeleton") for attaching additional language constructs such as metaclasses, metaassociations, and metaattributes. Although the Core package contains sufficient semantics to define the remainder of UML, it is not the UML meta-metamodel. It is the underlying base for the Foundation package, which in turn serves as the infrastructure for the rest of language. In other packages the Core is extended by adding metaclasses to the backbone using generalizations and associations.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Core package.

4.2 ABSTRACT SYNTAX

The abstract syntax for the Core package is expressed in graphic notation in Figure 5 and Figure 6. Figure 5 shows the model elements that form the structural backbone of the metamodel. Figure 6 shows the model elements that define relationships.

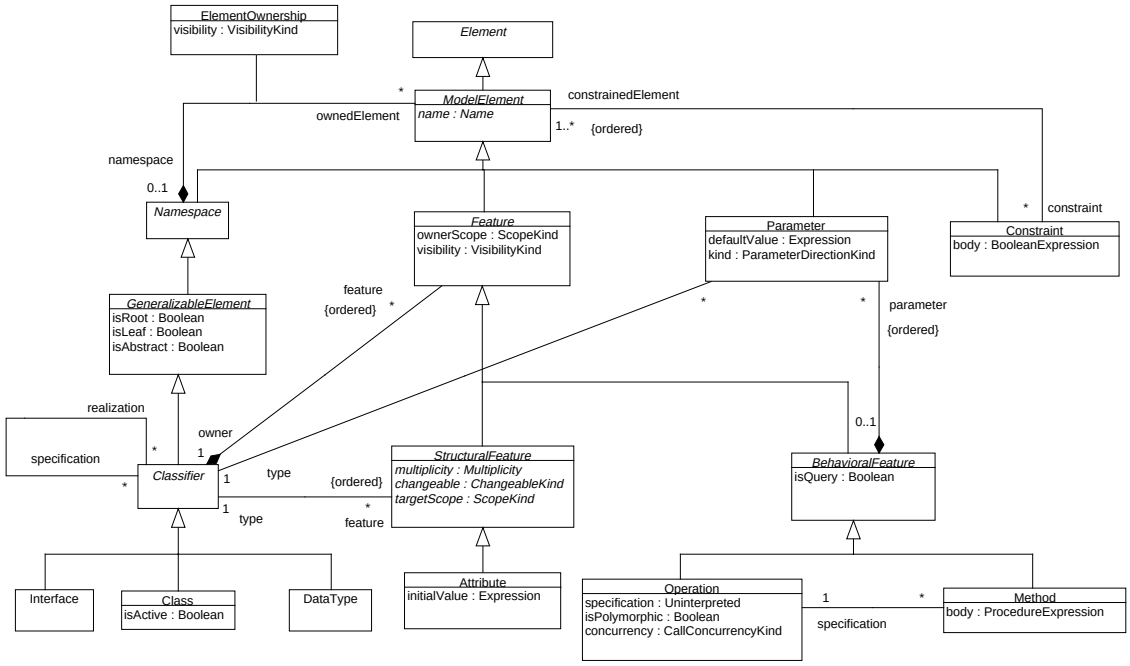


Figure 5: Core Package–Backbone

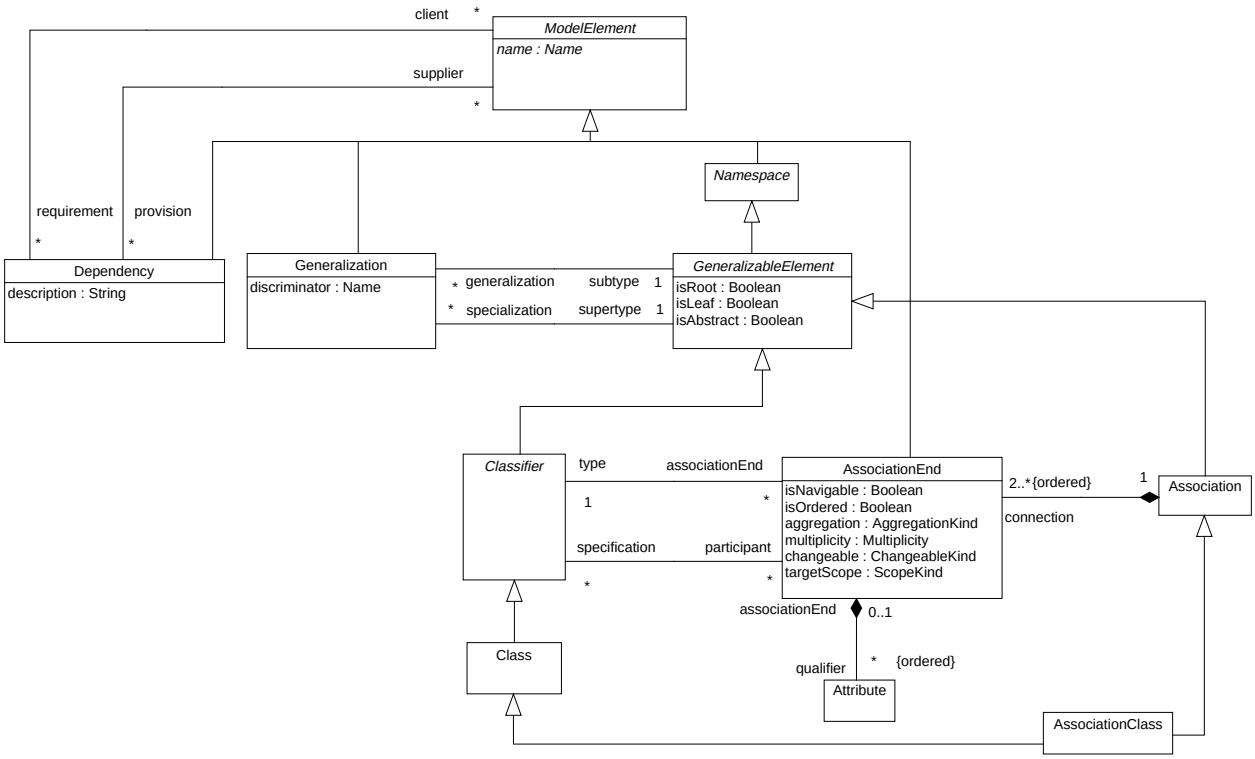


Figure 6: Core Package–Relationships

The following metaclasses are contained in the Core package:

Association

An *association* defines a semantic relationship between classifiers; the instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once.

In the metamodel an ***Association*** is a declaration of a semantic relationship between ***Classifiers***, such as ***Classes***. An ***Association*** has at least two ***AssociationEnds***. Each end is connected to a ***Classifier***—the same ***Classifier*** may be connected to more than one ***AssociationEnds*** in the same ***Association***. The ***Association*** represents a set of connections among instances of the ***Classifiers***. An instance of an ***Association*** is a ***Link***, which is a tuple of Instances drawn from the corresponding ***Classifiers***.

Attributes

name — The name of the ***Association*** which, in combination with its associated ***Classifiers***, must be unique within the enclosing namespace (usually a ***Package***).

Associations

connection — An ***Association*** consists of at least two ***AssociationEnds***, each of which represents a connection of the association to a ***Classifier***. Each ***AssociationEnd*** specifies a set of properties that must be fulfilled for the relationship to be valid. The bulk of the structure of an ***Association*** is defined by its ***AssociationEnds***.

AssociationClass

An *association class* is an association that is also a class. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not any of the classifiers.

In the metamodel an ***AssociationClass*** is a declaration of a semantic relationship between ***Classifiers*** which has a set of features of its own. ***AssociationClass*** is a subclass of both ***Association*** and ***Class*** (i.e. each ***AssociationClass*** is both an ***Association*** and a ***Class***). Therefore, an ***AssociationClass*** has both ***AssociationEnds*** and ***Features***.

AssociationEnd

An *association end* is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association; the association-ends of each association are ordered.

In the metamodel an ***AssociationEnd*** is part of an ***Association*** and specifies the connection of an ***Association*** to a ***Classifier***. It has a name and defines a set of properties of the connection**.** e.g., which ***Classifier*** the ***Instances*** must conform to, their multiplicity, and if they may be reached from another ***Instance*** via this connection.

In the following descriptions when referring to an association end for a binary association, the *source* end is the other end; the *target* end is the one whose properties are being discussed.

Attributes

aggregation — When placed on a target end, specifies whether the target end is an aggregation with respect to the source end. Only one end can be an aggregation. Possibilities are:

- none — The end is not an aggregate.
- aggregate — The end is an aggregate; the other end is therefore a part and must have the aggregation value of none. The part may be contained in other aggregates.
- composite — The end is a composite; the other end is therefore a part and must have the *aggregation* value of *none*. The part is strongly owned by the composite and may not be part of any other composite.

changeable — When placed on a target end, specifies whether an instance of the ***Association*** may be modified from the source end. Possibilities are:

- none — No restrictions on modification.
- frozen — No links may be added after the creation of the source object.
- addOnly — Links may be added at any time from the source object, but once created a link may not be removed before at least one participating object is destroyed.

isOrdered — When placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered. The ordering must be determined and maintained by ***Operations*** that add links; it represents additional information not inherent in the objects or links themselves. A set of ordered links can be scanned in order. The alternative is that the links form a set with no inherent ordering.

isNavigable When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. Specification of each direction across the ***Association*** is independent.

multiplicity When placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given ***Association***. (See ***Multiplicity***.)

name The role name of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifier, i.e. it may be used in the same way as an ***Attribute***, and must be unique with respect to ***Attributes*** and other pseudo-attributes of the source ***Classifier***.

targetScope Specifies whether the targets are ordinary ***Instances*** or are ***Classifiers***. Possibilities are:

instance Each link of the ***Association*** contains a reference to an ***Instance*** of the target ***Classifier***. This is the setting for a normal ***Association***.

classifier Each link of the ***Association*** contains a reference to the target ***Classifier*** itself. This represents a way to store meta-information.

Associations

qualifier An optional list of qualifier ***Attributes*** for the end. If the list is empty then the ***Association*** is not qualified.

specification Designates zero or more ***Classifiers*** that specify the ***Operations*** that may be applied to an ***Instance*** accessed by the ***AssociationEnd*** across the ***Association***. These determine the minimum interface that must be realized by the actual ***Classifier*** attached to the end to support the intent of the ***Association***. May be an ***Interface*** or another ***Classifier***.

type Designates the ***Classifier*** connected to the end of the ***Association***. It may not be an ***Interface*** because they have no physical structure.

Attribute

An *attribute* is a named slot within a classifier that describes a range of values that instances of the classifier may hold.

In the metamodel an ***Attribute*** is a named piece of the declared state of a ***Classifier,*** particularly the range of values that ***Instances*** of the ***Classifier*** may hold.

(The following list includes properties from ***StructuralFeature*** which has no other subclasses in the current metamodel.)

Attributes

changeable Whether the value may be modified after the object is created. Possibilities are:

none No restrictions on modification.

frozen	The value may not be altered after the object is instantiated and its values initialized. No additional values may be added to a set.
addOnly	(Meaningful only if the multiplicity is not fixed to a single value.) Additional values may be added to the set of values, but once created a value may not be removed or altered.
initialValue	An ***Expression*** specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supercede an initial value.)
multiplicity	The possible number of data values for the attribute that may be held by an instance. The cardinality of the set of values is an implicit part of the attribute. (See ***Multiplicity***.) In the common case in which the multiplicity is 1..1 then the attribute is a scalar, i.e., it holds exactly one value.

Associations

type	Designates the classifier whose instances are values of the attribute. Must be a ***Class*** or ***DataType***.

BehavioralFeature

A *behavioral feature* refers to a dynamic feature of a model element, such as an operation or method.

In the metamodel a ***BehavioralFeature*** specifies a behavioral aspect of a ***Classifier***. All different kinds of behavioral aspects of a ***Classifier***, such as ***Operation*** and ***Method***, are subclasses of ***BehavioralFeature***. ***BehavioralFeature*** is an abstract metaclass.

Attributes

isQuery	Specifies whether an execution of the ***Feature*** leaves the state of the system unchanged. True indicates that the state is unchanged; false indicates that side-effects may occur..
name	The name of the ***Feature***. The entire signature of the ***Feature*** (name and parameter list) must be unique within its containing ***Classifier***.

Associations

parameters	An ordered list of ***Parameters*** for the ***Operation***. To call the ***Operation***, the caller must supply a list of values compatible with the types of the ***Parameters***.

Class

A *class* is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

In the metamodel a ***Class*** describes a set of ***Objects*** sharing a collection of ***Features***, including ***Operations***, ***Attributes*** and ***Methods***, that are common to the set of ***Objects***. Furthermore, a ***Class*** may *realize* zero or more ***Interfaces***; this means that its *full descriptor* (see *Inheritance* for the definition) must contain every ***Operation*** from every realized ***Interface*** (it may contain additional

operations as well).

A ***Class*** defines the data structure of ***Objects***, although some ***Classes*** may be abstract, i.e. no ***Objects*** can be created directly from them. Each ***Object*** instantiated from a ***Class*** contains its own set of values corresponding to the ***StructuralFeatures*** declared in the *full descriptor*. ***Objects*** do not contain values corresponding to ***BehavioralFeatures*** or class-scope ***Attributes;*** all ***Objects*** of a ***Class*** share the definitions of the ***BehavioralFeatures*** from the ***Class***, and they all have access to the single value stored for each class-scope attribute.

Attributes

isActive	Specifies whether an ***Object*** of the ***Class*** maintains its own thread of control. If true, then an ***Object*** has its own thread of control and runs concurrently with other active ***Objects***. If false, then ***Operations*** run in the address space and under the control of the active ***Object*** that controls the caller.

Classifier

A *classifier* is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, and others that are defined in other metamodel packages.

In the metamodel a ***Classifier*** declares a collection of ***Features***, such as ***Attributes***, ***Methods***, and ***Operations***. It has a name, which is unique in the ***Namespace*** enclosing the ***Classifier***. ***Classifier*** is an abstract metaclass.

Associations

feature	A list of ***Features***, like ***Attribute***, ***Operation***, ***Method***, owned by the ***Classifier***.
participant	Inverse of *specification* on association to ***AssociationEnd.*** Denotes that the ***Classifier*** participates in an ***Association***.
realization	Inverse of *specification*. A set of ***Classifiers*** that implement the ***Operations*** of the ***Classifier***. These may not include ***Interfaces***.
specification	A set of ***Classifiers*** that specify the ***Operations*** that the ***Classifier*** must implement. The ***Classifier*** may implement more ***Operations*** than contained in the set of ***Classifiers***. The set may include ***Interfaces*** but is not restricted to them.

Constraint

A *constraint* is a semantic condition or restriction.

In the metamodel a ***Constraint*** is a ***BooleanExpression*** on an associated ***ModelElement***(s) which must be true for the model to be well formed. This restriction can be stated in natural language, or in different kinds of languages with a well-defined semantics. Certain ***Constraints*** are predefined in the UML, others may be user defined. Note that a ***Constraint*** is an assertion, not an executable mechanism; it indicates a restriction that must be enforced by correct design of a system.

Attributes

body	A ***BooleanExpression*** that must be true when evaluated for an instance of a system to be well-formed.

Associations

constrainedElement	A ***ModelElement*** or list of ***ModelElements*** affected by the ***Constraint***.

DataType

A *data type* is a type whose values have no identity, i.e. they are pure values. Data types include primitive built-in types (such as integer and string) as well as definable enumeration types (such as the predefined enumeration type boolean whose literals are false and true).

In the metamodel a ***DataType*** defines a special kind of type in which ***Operations*** are all pure functions, i.e. they can return ***DataValues*** but they cannot change ***DataValues*** (because they have no identity).

Dependency

A *dependency* states that the implementation or functioning of one or more elements require the presence of one or more other elements. All of the elements must exist at the same level of meaning, i.e., they do not involve a shift in the level of abstraction or realization.

In the metamodel a ***Dependency*** is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier, i.e. the client element requires the presence and knowledge of the supplier element.

Dependencies may be stereotyped to differentiate various kinds of dependency.

Attributes

description	A text description of the dependency.

Associations

client	The ***ModelElement*** or set of ***ModelElements*** that require the presence of the supplier.
supplier	The ***ModelElement*** or set of ***ModelElements*** whose presence is required by the client.

Element

An *element* is an atomic constituent of a model.

In the metamodel an ***Element*** is the top metaclass in the metaclass hierarchy. It has two subclasses: ***ModelElement*** and ***ViewElement***. ***Element*** is an abstract metaclass.

ElementOwnership

Element ownership has visibility in a namespace.

In the metamodel ***ElementOwnership*** reifies the relationship between ***ModelElement*** and ***Namespace*** denoting the ownership of a ***ModelElement*** by a ***Namespace*** and its visibility outside the ***Namespace***. See ***ModelElement***.

Feature

A *feature* is a property, like operation or attribute, which is encapsulated within another entity, such as an interface, a class, or a data type.

In the metamodel a ***Feature*** declares a behavioral or structural characteristic of an ***Instance*** of a ***Classifier*** or of the ***Classifier*** itself. ***Feature*** is an abstract metaclass.

Attributes

name — The name used to identify the ***Feature*** within the ***Classifier*** or ***Instance***. It must be unique across inheritance of names from ancestors including names of outgoing ***AssociationEnds***.

ownerScope — Specifies whether ***Feature*** appears in each ***Instance*** of the ***Classifier*** or whether there is just a single instance of the ***Feature*** for the entire ***Classifier***. Possibilities are:

- instance — Each ***Instance*** of the ***Classifier*** holds its own value for the ***Feature***.
- classifier — There is just one value of the ***Feature*** for the entire ***Classifier***.

visibility — Specifies whether the ***Feature*** can be used by other ***Classifier***. Visibilities of nested ***Namespaces*** combine so that the most restrictive visibility is the result. Possibilities:

- public — Any outside ***Classifier*** with visibility to the ***Classifier*** can use the ***Feature***.
- protected — Any descendent of the ***Classifier*** can use the ***Feature***.
- private — Only the ***Classifier*** itself can use the ***Feature***.

Associations

owner — The ***Classifier*** containing the ***Feature***.

GeneralizableElement

A *generalizable element* is a model element that may participate in a generalization relationship.

In the metamodel a ***GeneralizableElement*** can be a generalization of other ***GeneralizableElements***, i.e. all ***Features*** defined in and all ***ModelElements*** contained in the ancestors are also present in the ***GeneralizableElement***. ***GeneralizableElement*** is an abstract metaclass.

Attributes

isAbstract — Specifies whether the ***GeneralizableElement*** is an incomplete declaration or not. True indicates that the ***GeneralizableElement*** is an incomplete declaration (abstract), false indicates that it is complete (concrete). An abstract ***GeneralizableElement*** is not instantiable since it does not contain all necessary information.

isLeaf — Specifies whether the ***GeneralizableElement*** is an ***GeneralizableElement*** with no descendents. True indicates that it is and may not add descendents, false indicates that it may add descendents (whether or not it actually has any descendents at the moment).

isRoot Specifies whether the ***GeneralizableElement*** is a root ***GeneralizableElement*** with no ancestors. True indicates that it is and may not add ancestors; false indicates that it may add ancestors (whether or not it actually has any ancestors at the moment).

Associations

generalization Designates a ***Generalization*** whose **supertype** ***GeneralizableElement*** is the immediate ancestor of the current ***GeneralizableElement***.

specialization Designates a ***Generalization*** whose **subtype** ***GeneralizableElement*** is the immediate descendent of the current ***GeneralizableElement***.

Generalization

A *generalization* is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.

In the metamodel a ***Generalization*** is a directed inheritance relationship, uniting a ***GeneralizableElement*** with a more general ***GeneralizableElement*** in a hierarchy. ***Generalization*** is a subtyping relationship, i.e. an ***Instance*** of the more general ***GeneralizableElement*** may be substituted by an ***Instance*** of the more specific ***GeneralizableElement***. See *Inheritance* for the consequences of ***Generalization*** relationships.

Attributes

discriminator Designates the partition to which the ***Generalization*** link belongs. All of the ***Generalization*** links that share a given supertype ***GeneralizableElement*** are divided into groups by their discriminator names. Each group of links sharing a discriminator name represents an orthogonal dimension of specialization of the supertype ***GeneralizableElement***. The discriminator need *not* be unique. The empty string is considered just another name. If all of the ***Generalization*** below a given ***GeneralizableElement*** have the same name (including the empty name) then it is a plain set of subelements. Otherwise the subelements form two or more groups, each of which must be represented by one of its members as an ancestor in a concrete descendent element.

Associations

supertype Designates a ***GeneralizableElement*** that is the generalized version of the subtype ***GeneralizableElement***.

subtype Designates a ***GeneralizableElement*** that is the specialized version of the supertype ***GeneralizableElement***.

Interface

An *interface* is a declaration of a collection of operations that may be used for defining a service offered by an instance.

In the metamodel an ***Interface*** contains a set of ***Operations*** that together define a service offered by a ***Classifier*** realizing the ***Interface***. A ***Classifier*** may offer several services, which means that it may realize several ***Interfaces***, and several ***Classifiers*** may realize the same ***Interface***.

Interfaces are ***GeneralizableElements***. All ***Operations*** declared by an heir must either be new ***Operations*** or be specializations (restrictions) of ***Operations*** declared in its ancestor(s).

Interfaces may not have ***Attributes***, ***Associations***, or ***Methods***.

Method

A *method* is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.

In the metamodel a ***Method*** is a declaration of a named piece of behavior in a ***Classifier*** and realizes one or a set of ***Operations*** of the ***Classifier***.

Attributes

body	The implementation of the ***Method*** as a ***ProceduralExpression***.

Associations

specification	Designates an ***Operation*** that the ***Method*** implements. The ***Operation*** must be owned by the ***Classifier*** that owns the ***Method*** or be inherited by it. The signatures of the ***Operation*** and ***Method*** must match.

ModelElement

A *model element* is an element that is an abstraction drawn from the system being modeled. Contrast with *view element*, which is an element whose purpose is to provide a presentation of information for human comprehension.

In the metamodel a ***ModelElement*** is a named entity in a ***Model***. It is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either direct or indirect subclasses of ***ModelElement***. ***ModelElement*** is an abstract metaclass.

Attributes

name	An identifier for the ***ModelElement*** within its containing ***Namespace***.

Associations

constraint	A set of ***Constraints*** affecting the element.
provision	Inverse of ***supplier***. Designates a ***Dependency*** in which the ***ModelElement*** is a supplier.
requirement	Inverse of ***client.*** Designates a ***Dependency*** in which the ***ModelElement*** is a client.
namespace	Designates the ***Namespace*** that contains the ***ModelElement***. Every ***ModelElement*** except a root element must belong to exactly one ***Namespace***. The pathname of ***Namespace*** names starting from the system provides a unique designation for every ***ModelElement***. The association attribute ***visibility*** specifies the visibility of the element outside its namespace (see ***Visibility***).

Namespace

A *namespace* is a part of a model in which each name has a unique meaning.

In the metamodel a ***Namespace*** is a ***ModelElement*** that can own other ***ModelElements***, like ***Associations*** and ***Classifiers***. The name of each owned ***ModelElement*** must be unique within the ***Namespace***. Moreover, each contained ***ModelElement*** is owned by at most one ***Namespace.*** The concrete subclasses of ***Namespace*** have additional constraints on which kind of elements may be contained. ***Namespace*** is an abstract metaclass.

Associations

ownedElement A set of ***ModelElements*** owned by the ***Namespace***.

Operation

An *operation* is a service that can be requested from an object to effect behavior. An operation has a signature, which describes the actual parameters that are possible (including possible return values).

In the metamodel an ***Operation*** is a ***BehavioralFeature*** that can be applied to the ***Instances*** of the ***Classifier*** that contains the ***Operation***.

Attributes

concurrency Specifies the semantics of concurrent calls to the same passive instance, i.e. an ***Instance*** originating from a ***Classifier*** with *isActive=false*. (Active instances control access to their own ***Operations*** so this property is usually (although not required in UML) set to *sequential*.) Possibilities:

sequential Callers must coordinate so that only one call to an ***Instance*** (on any sequential ***Operation***) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.

guarded Multiple calls from concurrent threads may occur simultaneously to one ***Instance*** (on any guarded ***Operation***), but only one is allowed to commence; the others are blocked until the performance of the first ***Operation*** is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded ***Operations*** must perform correctly (or block themselves) in the case of a simultaneous sequential ***Operation*** or guarded semantics cannot be claimed.

concurrent Multiple calls from concurrent threads may occur simultaneously to one ***Instance*** (on any concurrent ***Operations***). All of them may proceed concurrently with correct semantics. Concurrent ***Operations*** must perform correct in the case of a simultaneous sequential or guarded ***Operation*** or concurrent semantics cannot be claimed.

isPolymorphic Whether the implementation of the ***Operation*** may be overridden by subclasses. If true, then ***Methods*** may be defined on subclasses. If false, then the ***Method*** realizing the ***Operation*** in the current ***Classifier*** is inherited unchanged by all descendents.

specification Description of the effects of performing an ***Operation***, stated as an ***Expression***.

Parameter

A *parameter* is an unbound variable that can be changed, passed or returned. A parameter may include a name, type and direction of communication. Parameters are used in the specification of operations, messages and events, templates etc.

In the metamodel a ***Parameter*** is a declaration of an argument to be passed to, or returned from an ***Operation***, a ***Signal***, etc.

Attributes

defaultValue	An ***Expression*** whose evaluation yields a value to be used when no argument is supplied for the ***Parameter***.
kind	Specifies what kind of a ***Parameter*** is required. Possibilities are:
in	An input ***Parameter*** (may not be modified)
out	An output ***Parameter*** (may be modified to communicate information to the caller).
inout	An input ***Parameter*** that may be modified.
return	A return value of a call.
name	The name of the ***Parameter***, which must be unique within its containing ***Parameter*** list.

Associations

type	Designates a ***Classifier*** to which an argument value must conform.

StructuralFeature

A *structural feature* refers to a static feature of a model element, such as an attribute.

In the metamodel a ***StructuralFeature*** declares a structural aspect of an ***Instance*** of a ***Classifier***, such as an ***Attribute***. It specifies e.g. the multiplicity and changeability of the ***StructuralFeature***. ***StructuralFeature*** is an abstract metaclass.

See ***Attribute*** for the descriptions of the attributes and associations, as it is the only subclass of ***StructuralFeature*** in the current metamodel.

4.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Core package.

Association

[1] The ***AssociationEnds*** must have a unique name within the ***Association***.

```
self.allConnections->forAll( r1, r2 | r1.name = r2.name implies r1 = r2 )
```

[2] At most one ***AssociationEnd*** may be an aggregation or composition.

```
self.allConnections->select(aggregation <> #none)->size <= 1
```

[3] If an ***Association*** has 3 or more ***AssociationEnds*** then no ***AssociationEnd*** may be an aggregation or composition.

[4] The connected ***Classifiers*** of the ***AssociationEnds*** should be included in the ***Namespace*** of the ***Association***.

```
self.allConnections->forAll (r |
   self.namespace.allContents->includes (r.type) )
```

Additional operations

[1] The operation *allConnections* results in the set of all ***AssociationEnds*** of the ***Association***.

```
allConnections : Set(AssociationEnd);
allConnections = self.connection
```

AssociationClass

[1] The names of the ***AssociationEnds*** and the ***StructuralFeatures*** do not overlap.

```
self.allConnections->forAll( ar |
   self.allFeatures->forAll( f |
      f.oclIsKindOf(StructuralFeature) implies ar.name <> f.name ))
```

[2] An ***AssociationClass*** cannot be defined between itself and something else.

```
self.allConnections->forAll(ar | ar.type <> self)
```

Additional operations

[1] The operation *allConnections* results in the set of all ***AssociationEnds*** of the ***AssociationClass***, including all connections defined by its supertype (transitive closure).

```
allConnections : Set(AssociationEnd);
allConnections = self.connection->union(self.supertype->select
   (s | s.oclIsKindOf(Association))->collect (a : Association |
      a.allConnections))->asSet
```

AssociationEnd

[1] The ***Classifier*** of an ***AssociationEnd*** cannot be an ***Interface*** or a ***DataType*** unless the ***DataType*** is part of a composite aggregation.

```
not self.type.oclIsKindOf (Interface)
and
( self.type.oclIsKindOf (DataType) implies
   self.association.connection->select ( ae | ae <> self)->forAll ( ae |
      ae.aggregation = #composite) )
```

[2] An ***Instance*** may not belong by composition to more than one composite ***Instance***.

```
self.aggregation = #composite implies self.multiplicity.max <= 1
```

Attribute

No extra well-formedness rules.

BehavioralFeature

[1] All ***Parameters*** should have a unique name.

```
self.parameter->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)
```

[2] The type of the ***Parameters*** should be included in the ***Namespace*** of the ***Classifier***.

```
self.parameter->forAll( p |
   self.owner.namespace.allContents->includes (p.type) )
```

Additional operations

[1] The operation *hasSameSignature* checks if the argument has the same signature as the instance itself.

```
hasSameSignature ( b : BehavioralFeature ) : Boolean;
hasSameSignature (b) =
   (self.name = b.name) and
   (self.parameter->size = b.parameter->size) and
   Sequence{ 1..(self.parameter->size) }->forAll( index : Integer |
      b.parameter->at(index).type =
         self.parameter->at(index).type and
      b.parameter->at(index).kind =
         self.parameter->at(index).kind
   )
```

Class

[1] If a ***Class*** is concrete, all the ***Operations*** of the ***Class*** should have a realizing ***Method*** in the *full descriptor*.

```
not self.isAbstract implies self.allOperations->forAll (op |
self.allMethods->exists (m | m.specification->includes(op)))
```

[2] A ***Class*** can only contain ***Classes***, ***Associations***, ***Generalizations***, ***UseCases, Constraints***, ***Dependencies, Collaborations,*** and ***Interfaces*** as a ***Namespace***.

```
self.allContents->forAll->(c |
   c.oclIsKindOf(Class         ) or
   c.oclIsKindOf(Association   ) or
   c.oclIsKindOf(Generalization) or
   c.oclIsKindOf(UseCase       ) or
   c.oclIsKindOf(Constraint    ) or
   c.oclIsKindOf(Dependency    ) or
   c.oclIsKindOf(Collaboration ) or
   c.oclIsKindOf(Interface     )
)
```

[3] For each ***Operation*** in an ***Interface*** provided by the ***Class,*** the ***Class*** must have a matching ***Operation***.

```
self.specification.allOperations->forAll (interOp |
   self.allOperations->exists( op | op.hasSameSignature (interOp) ) )
```

Classifier

[1] No ***BehavioralFeature*** of the same kind may have the same signature in a ***Classifier***.

```
self.feature->forAll(f, g |
   (
      (
         (f.oclIsKindOf(Operation) and g.oclIsKindOf(Operation)) or
         (f.oclIsKindOf(Method   ) and g.oclIsKindOf(Method   )) or
         (f.oclIsKindOf(Reception) and g.oclIsKindOf(Reception))
      ) and
      f.oclAsType(BehavioralFeature).hasSameSignature(g)
   )
implies f = g)
```

[2] No ***Attributes*** may have the same name within a ***Classifier***.

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( p, q |
   p.name = q.name implies p = q )
```

[3] No opposite ***AssociationEnds*** may have the same name within a ***Classifier***.

```
self.oppositeEnds->forAll ( p, q | p.name = q.name implies p = q )
```

[4] The name of an ***Attribute*** may not be the same as the name of an opposite ***AssociationEnd*** or a ***ModelElement*** contained in the ***Classifier***.

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( a |
   not self.allOppositeAssociationEnds->union (self.allContents)->collect ( q |
      q.name )->includes (a.name) )
```

[5] The name of an opposite ***AssociationEnd*** may not be the same as the name of an ***Attribute*** or a ***ModelElement*** contained in the ***Classifier***.

```
self.oppositeAssociationEnds->forAll ( o |
   not self.allAttributes->union (self.allContents)->collect ( q |
      q.name )->includes (o.name) )
```

Additional operations

[1] The operation *allFeatures* results in a Set containing all ***Features*** of the ***Classifier*** itself and all its inherited ***Features***.

```
allFeatures : Set(Feature);
allFeatures = self.feature->union(
                        self.supertype.oclAsType(Classifier).allFeatures)
```

[2] The operation *allOperations* results in a Set containing all ***Operations*** of the ***Classifier*** itself and all its inherited ***Operations***.

```
allOperations : Set(Operation);
allOperations = self.allFeatures->select(f | f.oclIsKindOf(Operation))
```

[3] The operation *allMethods* results in a Set containing all ***Methods*** of the ***Classifier*** itself and all its inherited ***Methods***.

```
allMethods : set(Method);
allMethods = self.allFeatures->select(f | f.oclIsKindOf(Method))
```

[4] The operation *allAttributes* results in a Set containing all ***Attributes*** of the ***Classifier*** itself and all its inherited ***Attributes***.

```
allAttributes : set(Attribute);
allAttributes = self.allFeatures->select(f | f.oclIsKindOf(Attribute))
```

[5] The operation *associations* results in a Set containing all ***Associations*** of the ***Classifier*** itself.

```
associations : set(Association);
associations = self.associationEnd.association->asSet
```

[6] The operation *allAssociations* results in a Set containing all ***Associations*** of the ***Classifier*** itself and all its inherited ***Associations***.

```
allAssociations : set(Association);
allAssociations = self.associations->union (
                  self.supertype.oclAsType(Classifier).allAssociations)
```

[7] The operation *oppositeAssociationEnds* results in a set of all ***AssociationEnds*** that are opposite to the ***Classifier***.

```
oppositeAssociationEnds : Set (AssociationEnd);
oppositeAssociationEnds =
   self.association->select ( a | a.associationEnd->select ( ae |
      ae.type = self ).size = 1 )->collect ( a |
         a.associationEnd->select ( ae | ae.type <> self ) )->union (
   self.association->select ( a | a.associationEnd->select ( ae |
      ae.type = self ).size > 1 )->collect ( a |
```

```
        a.associationEnd) )
```

[8] The operation *allOppositeAssociationEnds* results in a set of all ***AssociationEnds***, including the inherited ones, that are opposite to the ***Classifier***.

```
allOppositeAssociationEnds : Set (AssociationEnd);
allOppositeAssociationEnds = self.oppositeAssociationEnds->union (
      self.supertype.allOppositeAssociationEnds )
```

Constraint

[1] A ***Constraint*** cannot be applied to itself.

```
not self.constrainedElement->includes (self)
```

DataType

[1] A ***DataType*** can only contain ***Operations***, which all must be queries.

```
self.allFeatures->forAll(f |
                f.oclIsKindOf(Operation) and f.oclAsType(Operation).isQuery)
```

[2] A ***DataType*** cannot contain any other ***ModelElements***.

```
self.allContents->isEmpty
```

Dependency

No extra well-formedness rules.

Element

No extra well-formedness rules.

ElementOwnership

No additional well-formedness rules.

Feature

No extra well-formedness rules.

GeneralizableElement

[1] A root cannot have any ***Generalizations***.

```
self.isRoot implies self.generalization->isEmpty
```

[2] No ***GeneralizableElement*** can have a supertype ***Generalization*** to an element which is a leaf.

```
self.supertype->forAll(s | not s.isLeaf)
```

[3] Circular inheritance is not allowed.

```
not self.allSupertypes->includes(self)
```

[4] The supertype must be included in the ***Namespace*** of the ***GeneralizableElement***.

```
self.generalization->forAll(g |
  self.namespace.allContents->includes(g.supertype) )
```

Additional Operations

[1] The operation *allContents* returns a Set containing all ***ModelElements*** contained in the ***GeneralizableElement*** together with the contents inherited from its supertypes.

```
allContents : Set(ModelElement);
allContents = self.contents->union(
   self.supertype.allContents->select(e |
      e.elementOwnership.visibility = #public or
      e.elementOwnership.visibility = #protected))
```

[2] The operation *supertype* returns a Set containing all direct supertypes.

```
supertype : Set(GeneralizableElement);
supertype = self.generalization.supertype
```

[3] The operation *allSupertypes* returns a Set containing all the ***GeneralizableElements*** inherited by this ***GeneralizableElements*** (the transitive closure), excluding the ***GeneralizableElement*** itself.

```
allSupertypes : Set(GeneralizableElement);
allSupertypes = self.supertype->union(self.supertype.allSupertypes)
```

Generalization

[1] A ***GeneralizableElement*** may only be a subclass of ***GeneralizableElement*** of the same kind.

```
self.subtype.oclType = self.supertype.oclType
```

Interface

[1] An ***Interface*** can only contain ***Operations***.

```
self.allFeatures->forAll(f | f.oclIsKindOf(Operation))
```

[2] An ***Interface*** cannot contain any ***Classifiers.***

```
self.allContents->isEmpty
```

[3] All ***Features*** defined in an ***Interface*** are public***.***

```
self.allFeatures->forAll ( f | f.visibility = #public )
```

Method

[1] If one of the realized ***Operations*** is a query, then so is also the ***Method***.

```
self.specification->exists ( op | op.isQuery ) implies self.isQuery
```

[2] The signature of the ***Method*** should be the same as the signature of the realized ***Operations***.

```
self. specification->forAll ( op | self.hasSameSignature (op) )
```

[3] The visibility of the ***Method*** should be the same as for the realized ***Operations***.

```
self. specification->forAll ( op | self.visibility = op.visibility )
```

ModelElement

Additional Operations

[1] The operation *supplier* results in a Set containing all direct suppliers of the ***ModelElement***.

```
supplier : Set(ModelElement);
```

```
supplier = self.provision.supplier
```

[2] The operation *allSuppliers* results in a Set containing all the ***ModelElements*** that are suppliers of this ***ModelElement***, including the suppliers of these ***ModelElements***. This is the transitive closure.

```
allSuppliers : Set(ModelElement);
allSuppliers = self.supplier->union(self.supplier.allSuppliers)
```

[3] The operation *model* results in the ***Model*** to which a ***ModelElement*** belongs.

```
model : Set(Model);
model = self.namespace->union(self.namespace.allSurroundingNamespaces)
         ->select( ns|
               ns.oclIsKindOf (Model))
```

Namespace

[1] If a contained element, which is not an ***Association*** or ***Generalization***, has a name then the name must be unique in the ***Namespace***.

```
self.allContents->forAll(me1, me2 : ModelElement |
   (  not me1.oclIsKindOf (Association) and not me2.oclIsKindOf (Association) and
      me1.name <> '' and me2.name <> '' and me1.name = me2.name
   ) implies
      me1 = me2 )
```

[2] All ***Associations*** must have a unique combination of name and associated ***Classifiers*** in the ***Namespace***.

```
self.allContents->select(oclIsKindOf(Association))->
   forAll(a1, a2 : Association |
      (  a1.name = a2.name and
         a1.connection->size = a2.connection->size and
         Sequence{1..a1.connection->size}->forAll(i |
            a1.connection->at(i).type = a2.connection->at(i).type)
      ) implies
         a1 = a2)
```

Additional operations

[1] The operation *contents* results in a Set containing all ***ModelElements*** contained by the ***Namespace***.

```
contents : Set(ModelElement)
contents = self.ownedElement
```

[2] The operation *allContents* results in a Set containing all ***ModelElements*** contained by the ***Namespace***.

```
allContents : Set(ModelElement);
allContents = self.contents
```

[3] The operation *allVisibleElements* results in a Set containing all ***ModelElements*** visible outside of the ***Namespace***.

```
allVisibleElements : Set(ModelElement)
allVisibleElements = self.allContents->select(e |
   e.elementOwnership.visibility = #public)
```

[4] The operation *allSurroundingNamespaces* results in a Set containing all surrounding ***Namespaces***.

```
allSurroundingNamespaces : Set(Namespace)
allSurroundingNamespaces =
self.namespace->union(self.namespace.allSurroundingNamespaces)
```

Operation

No additional well-formedness rules.

Parameter

[1] An ***Interface*** cannot be used as the type of a parameter.

```
not self.type.oclIsKindOf(Interface)
```

StructuralFeature

[1] The connected type should be included in the current ***Namespace***.

```
self.owner.namespace.allContents->includes(self.type)
```

4.4 SEMANTICS

This section provides a description of the dynamic semantics of the elements in the Core. It is structured based on the major constructs in the core, such as interface, class, and association.

Inheritance

To understand *inheritance* it is first necessary to understand the concept of a *full descriptor* and a *segment descriptor*. A *full descriptor* is the full description needed to describe an object or other instance (see *Instantiation*). It contains a description of all of the attributes, associations, and operations that the object contains. In a pre-object-oriented language the full descriptor of a data structure was declared directly in its entirety. In an object-oriented language the description of an object is built out of incremental *segments* that are combined using *inheritance* to produce a full descriptor for an object. The segments are the modeling elements that are actually declared in a model; they include elements such as class and other generalizable elements. Each generalizable element contains a list of features and other relationships that it *adds* to what it *inherits* from its *ancestors*. The mechanism of *inheritance* defines how full descriptors are produced from a set of segments connected by generalization. The full descriptors are implicit, but they define the structure of actual instances.

Each kind of generalizable element has a set of *inheritable features*. For any model element, these include constraints. For classifiers**,** these include features (attributes, operations, signal takers, and methods) and participation in associations. The *ancestors* of a generalizable element are its supertypes (if any) together with all of their ancestors (with duplicates removed).

If a generalizable element has no supertype, then its full descriptor is the same as its segment descriptor. If a generalizable element has one or more supertypes, then its full descriptor contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may not be redefined). A method may be declared in more than one segment; a method declared in any segment supercedes and replaces a method with the same signature declared in any ancestor. If two or more methods nevertheless remain, then they *conflict* and the model is ill-formed. The constraints on the full descriptor are the union of the constraints on the segment itself and all of its ancestors; if any of them are inconsistent then the model is ill-formed.

In any full descriptor for a classifier, each method must have a corresponding operation. In a concrete classifier, each operation in its full descriptor must have a corresponding method in the full descriptor.

The purpose of the full descriptor is explained under *Instantiation.*

Instantiation

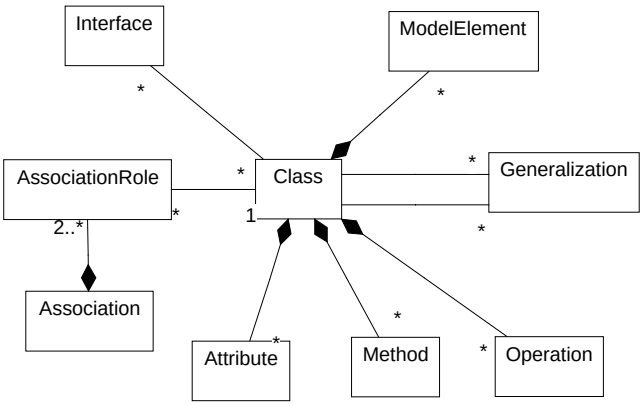
The purpose of a model is to describe the possible states of a system and their behavior. The state of a system comprises objects, values, and links. Each object is described by a full class descriptor; the class corresponding to this descriptor is the *direct class* of the object. Similarly each link has a *direct association* and each value has a *direct data type.* Each of these instances is said to be a *direct instance of* the classifier from which its full descriptor was derived. An instance is an *indirect instance of* the classifier or any of its ancestors.

The data content of an object comprises one value for each attribute in its full class descriptor (and nothing more); the value must consistent with the type of the attribute. The data content of a link comprises a tuple containing a list of instances, one that is an indirect instance of each participant classifier in the full association descriptor. The instances and links must obey any constraints on the full descriptors of which they are instances (including both explicit constraints and built-in constraints such as multiplicity).

The state of a system is a *valid system instance* if every instance in it is a direct instance of some element in the system model and if all of the constraints imposed by the model are satisfied by the instances.

The behavioral parts of UML describe the valid sequences of valid system instances that may occur as a result of both external and internal behavioral effects.

Class



The purpose of a *class* is to declare a collection of methods, operations, and attributes that fully describe the structure and behavior of objects. All objects instantiated from a class will have attribute values matching the attributes of the full class descriptor and support the operations found in the full class descriptor. Some classes may not be directly instantiated. These classes are said to be *abstract* and exist only for other classes to inherit and reuse the features declared by them; no object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is non-abstract.

When a class is instantiated to create a new object, a new instance is created , which is initialized containing an attribute value for each attribute found in the full class descriptor. The object is also initialized with a connection to the list of methods in the full class descriptor. (NB: An actual implementation behaves *as if* there were a full class descriptor, but many clever optimizations are

possible in practice.) Finally, the identity of the new object is returned to the creator. The identity of every instance in a well-formed system is unique and automatic.

A class can have *generalizations* to other classes. This means that the full class descriptor of a class is derived by inheritance from its own segment declaration and those of its ancestors. Generalization between classes implies substitutability, i.e. an instance of a class may be used whenever an instance of a superclass is expected. If the class is specified as a root it cannot be a subclass of other classes. Similarly, if it is specified as a leaf no other class can be a subclass of the class.

Each *attribute* declared in a class has a visibility and a type. The visibility defines if the attribute is publicly available to any class, if it is only available inside the class and its subclasses (*protected*), or if it can only be used inside the class (*private*). The *targetScope* of the attribute declares whether its value should be an instance (of a subtype) of that type or if it should be (a subtype of) the type itself. There are two alternatives for the *ownerScope* of an attribute: it may state that each object created by the class (or by its subclasses) has its own value of the attribute, or that the value is owned by the class itself. An attribute also declares how many attribute values should be connected to each owner (*multiplicity*), what the initial values should be, and if these attribute values may be changed: no constraints exists (*none*), the value cannot be replaced or added to once it has been initialized (*frozen*), or new values may be added to a set but not removed or altered (*addOnly*).

For each *operation* the operation name, the types of the parameters and the return type(s) are specified, as well as its visibility (see above). An operation may also include a specification of the effects of its invocation. The specification can be done in several different ways, e.g. with pre- and post-conditions, pseudo-code, or just plain text. Each operation declares if it is applicable to the instances the class or to the class itself (*ownerScope*). Furthermore, the operation states whether or not its application will modify the state of the object (*isQuery*). The operation also states whether or not the operation may be realized by a different method in a subclass (*isPolymorphic*). An operation may have a set of *extension points* specifying where additional behavior may be inserted into the operation. A method realizing an operation has the same signature as the operation and a body implementing the specification of the operation. Methods in descendents override and replace methods inherited from ancestors (see *Inheritance).* Each method implements an operation declared in the class or inherited from an ancestor; the same operation may not be declared more than once in a full class descriptor. The specification of the method must match the specification of its matching operation, as defined above for operations. Furthermore, if the isQuery attribute of an operation is true then it must also be true in any realizing method. However, if it is false in the operation it may still be true in the method (*isQuery=false*) does not require that the operation modify the state The concept of visibility is not relevant for methods.

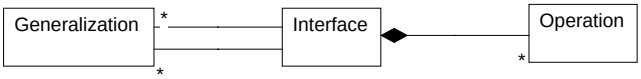
Classes may have *associations* to each other. This implies that objects created by the associated classes are semantically connected, i.e. that links exist between the objects, according to the requirements of the associations. See Association below. Associations are inherited by subclasses.

A class may realize a set of interfaces. This means that each operation found in the full descriptor for any realized interface must be present in the full class descriptor with the same specification. The relationship between interface and class is not necessarily one-to-one; a class may offer several interfaces and one interface may be offered by more than one class. The same operation may be defined in multiple interfaces that a class supports; if their specifications are identical then there is no conflict, otherwise the model is ill-formed. Moreover, a class may contain additional operations besides those found in its interfaces.

A class acts as the *namespace* for attributes, outgoing role names on associations, and operations.

Furthermore, since a class acts as a namespace for contained classes, interfaces, and associations (elements defined within its scope; they do not imply aggregation). The contained classifiers can be used as ordinary classifiers in the container class. However, the contents cannot be referenced by anyone outside the container class. If a class inherits another class the visibility of the contents as it is defined in the superclass guides if the contained elements are visible in the subclass. If the visibility of an element is *public* or *protected* then it is also visible in the subclass, but if the visibility is *private* then the element is not visible, and therefor not available, in the subclass.

Interface

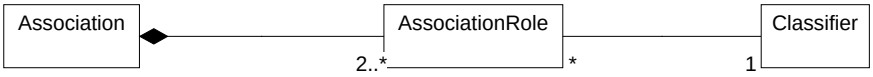


The purpose of an *interface* is to collect a set of operations that constitute a coherent service offered by classifiers. Interfaces provide a way to partition and characterize groups of operations. An interface is only a collection of operations with a name; it cannot be directly instantiated. Instantiable classifiers, such as class or use case, may use interfaces for specifying different services offered by their instances. Several classifiers may realize the same interface; all of them must contain at least the operations matching those contained in the interface. The specification of an *operation* contains the signature of the operation, i.e. its name, the types of the parameters and the return type. An interface does not imply any internal structure of the realizing classifier; for example, it does not define which algorithm to use for realizing an operation. An operation may, however, include a specification of the effects of its invocation. The specification can be done in several different ways, e.g. with pre- and post-conditions, pseudo-code, or just plain text.

Each operation declares if it applies to the instances of the classifier declaring it or to the classifier itself, e.g. a constructor on a class (*ownerScope*). Furthermore, the operation states whether or not its application will modify the state of the instance (*isQuery*). The operation also states whether or not all the classes must have the same realization of the operation (*isPolymorphic*).

An interface can be a subtype of other interfaces denoted by *generalizations*. This means that the a classifier offering the interface must provide not only the operations declared in the interface but also those declared in the ancestors of the interface. If the interface is specified as a root it cannot be a subtype of other interfaces. Similarly, if it is specified as a leaf no other interface can be a subtype of the interface.

Association



An *association* declares a connection (*link*) between instances of the associated classifiers, e.g. classes. It consists of at least two *association-ends*, each specifying a connected classifier and a set of properties which must be fulfilled for the relationship to be valid. The *multiplicity* property of an association-end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A *multiplicity* is a range of nonnegative integers. The association-end also states whether or not the connection may be traversed towards the instance playing that role in the connection (*isNavigable*), i.e. if the instance is directly reachable via the association. An association-end also specifies whether or not an instance playing that role in a connection may be replaced by another instance. It may state that no constraints exists (*none*), that the link cannot be modified once it has been initialized (*frozen*), or that new links of the association may be added but not removed or altered

(*addOnly*); these constraints do not affect the modifiability of the objects themselves that are attached to the links. Moreover, the *targetScope* specifies if the association-end should be connected to an instance of (a subtype of) the classifier or (a subtype of) the classifier itself. The *isOrdered* attribute of association-end states if the instances related to a single instance at the other end have an ordering that must be preserved. The order of insertion of new links must be specified by operations that add or modify links. Note that sorting is a performance optimization and is *not* an example of a logically ordered association, because the ordering information in a sort does not add any information.

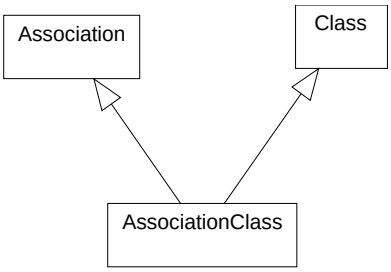
An association may represent an *aggregation*, i.e. a whole/part relationship. In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. *Composite* aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time, although the owner may be changed over time. Furthermore, a composite implies propagation semantics, i.e. some of the dynamic semantics of the whole is propagated to its parts. For example, if the whole is copied or deleted then so are the parts as well. A *shared* aggregation denotes weak ownership, i.e. the part may be included in several aggregates, and its owner may also change over time. However, the semantics of a shared aggregation does not imply deletion of the parts when the one of its containers is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship, i.e. the instances form a directed, non-cyclic graph. Composition instances form a strict tree (or rather a forest).

A *qualifier* declares a partition of the set of associated instances with respect to an instance at the qualified end (the *qualified instance* is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and therefore designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets each selected by a given qualifier instance. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences; in the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilities easy access of sets of associated instances linked by a given qualifier value.

Note that the multiplicity of a qualifier is given assuming that the qualifier value is supplied. The “raw” multiplicity without the qualifier is assumed to be 0..*. This is not fully general but it is almost always adequate, as a situation in which the raw multiplicity is 1 would best be modeled without a qualifier.

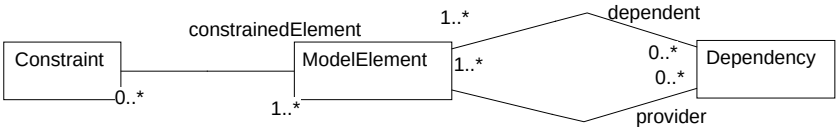
Note also that a qualified multiplicity whose lower bound is zero indicates that a given qualifier value may be absent, while a lower bound of 1 indicates that any possible qualifier value must be present. The latter is reasonable only for qualifiers with a finite number of values (such as enumerated values or integer ranges) that represent full tables indexed by some finite range of values.

AssociationClass



An association may be refined to have its own set of features, i.e. features that do not belong to any of the connected classifiers, but rather to the association itself. Such an association is called an *association class*. It will be both an association, connecting a set of classifiers, and a class, and as such have features and be included in other associations. The semantics of such an association is a combination of the semantics of an ordinary association and of a class.

Miscellaneous



A *constraint* is a Boolean expression over one or several elements which must always be true. A constraint can be specified in several different ways, e.g. using natural language or a constraint language.

A *dependency* specifies that the semantics of a set of model elements requires the presence of another set of model elements. This implies that if the source is somehow modified the dependents must probably be modified. The reason for the dependency can be specified in several different ways, e.g. using natural language or an algorithm, but is often implicit.

A special kind of classifier, which is similar to class, is *data type*, but the instances of a data type are primitive values, i.e. non-objects. For example, the integers and strings are usually treated as primitive values. A primitive value does not have an identity, so two occurrences of the same value cannot be differentiated. It is usually used for specification of the type of an attribute. An *enumeration type* is a user-definable type comprising a finite number of values.

4.5 STANDARD ELEMENTS

The predefined stereotypes, constraints and tagged values for the Core package are listed in Table 2 and defined in the *Standard Elements* appendix.

Table 2: Core—Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Association		implicit or	
Attribute			persistence
BehavioralFeature	«create» «destroy»		
Class	«implementationClass» «inherits» «type»		
Classifier	«process» «stereotype» «utility»		location persistence responsibility semantics
Constraints	«invariant» «metaclass» «postcondition» «powertype» «precondition»		
Element			documentation
Generalization	«private» «subclass» «subtype» «thread» «uses»	complete disjoint incomplete overlapping	
Operation			semantics

4.6 NOTES

In UML ***Associations*** can be of three different kinds: ordinary association, composite aggregate, and shared aggregate. Since the aggregate construct can have several different meanings depending on the application area, UML gives a more precise meaning to two of these constructs, i.e. association and composite aggregate, and leaves the shared aggregate more loosely defined in between.

Operation is a conceptual construct while ***Method*** is the implementation construct. Their common features, such as having a signature, are expressed in the ***BehavioralFeature*** metaclass, and the specific semantics of the ***Operation*** and the ***Method*** constructs are defined in the corresponding subclasses of ***BehavioralFeature***.

A ***Usage*** or ***Binding*** dependency can only be established between elements in the same model, since the semantics of a model cannot be dependent on the semantics of another model. If a connection is to be established between elements in different models a ***Trace*** or ***Refinement*** should be used.

The ***AssociationClass*** construct can be expressed in a few different ways in the metamodel, e.g. as a subclass of ***Class***, as a subclass of ***Association***, or as a subclass of ***Classifier***. Since an ***AssociationClass*** is a construct being both an association, having a set of association-ends, and a class, declaring a set of features. The most accurate way of expressing it is as a subclass of both ***Association*** and ***Class***. In this way ***AssociationClass*** will have all the properties of the other two constructs. Moreover, if new kinds of associations containing features (e.g. ***AssociationDataType***) are to be included in UML, these are easily added as subclasses of ***Association*** and the other ***Classifier***.

The two terms *subtype* and *subclass* are synonyms and mean that an instance of a classifier being a subtype of another classifier can always be used where an instance of the latter classifier is expected.

5. FOUNDATION PACKAGE: AUXILIARY ELEMENTS

5.1 OVERVIEW

The Auxiliary Elements package is the subpackage of the Foundation package that defines additional constructs that extend the Core. Auxiliary elements provide infrastructure for dependencies, templates, physical structures and view elements.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Auxiliary Elements package.

5.2 ABSTRACT SYNTAX

The abstract syntax for the Auxiliary Elements package is expressed in graphic notation in Figure 7 and Figure 8. Figure 7 shows the model elements that define dependencies and templates. Figure 8 shows the model elements that define physical structures and view elements.

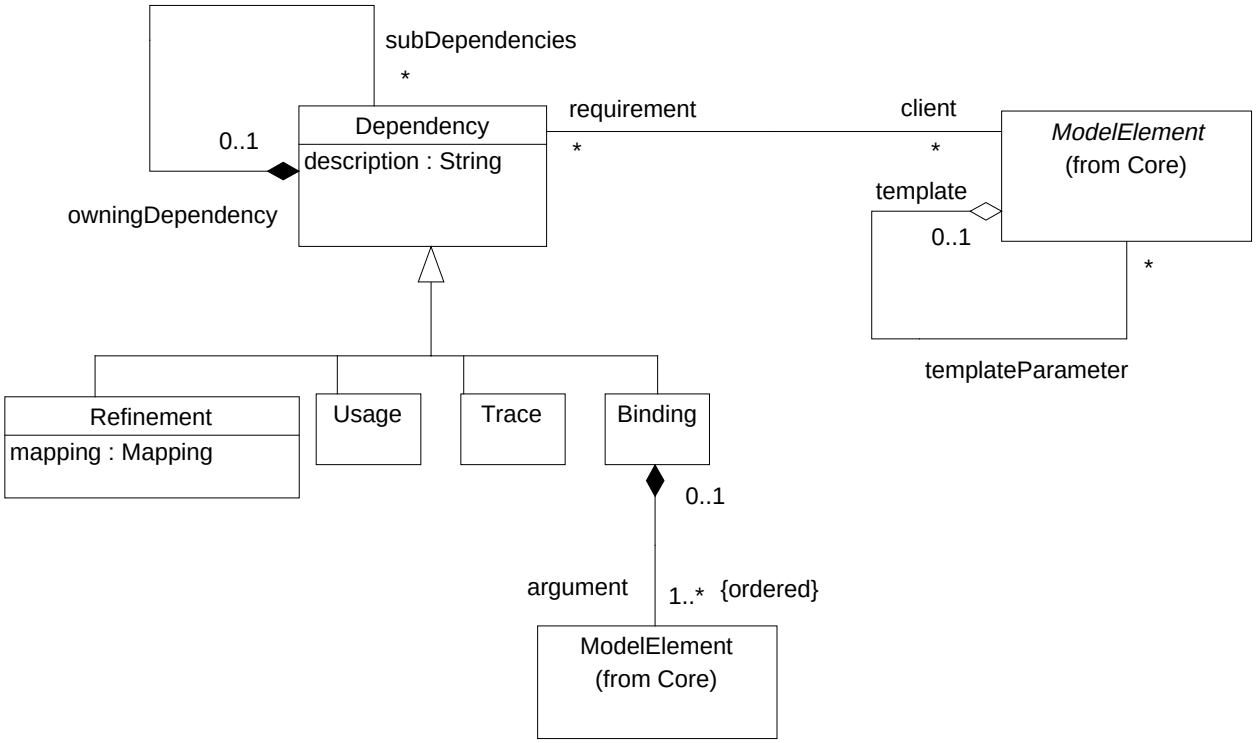


Figure 7: Auxiliary Elements—Dependencies and Templates

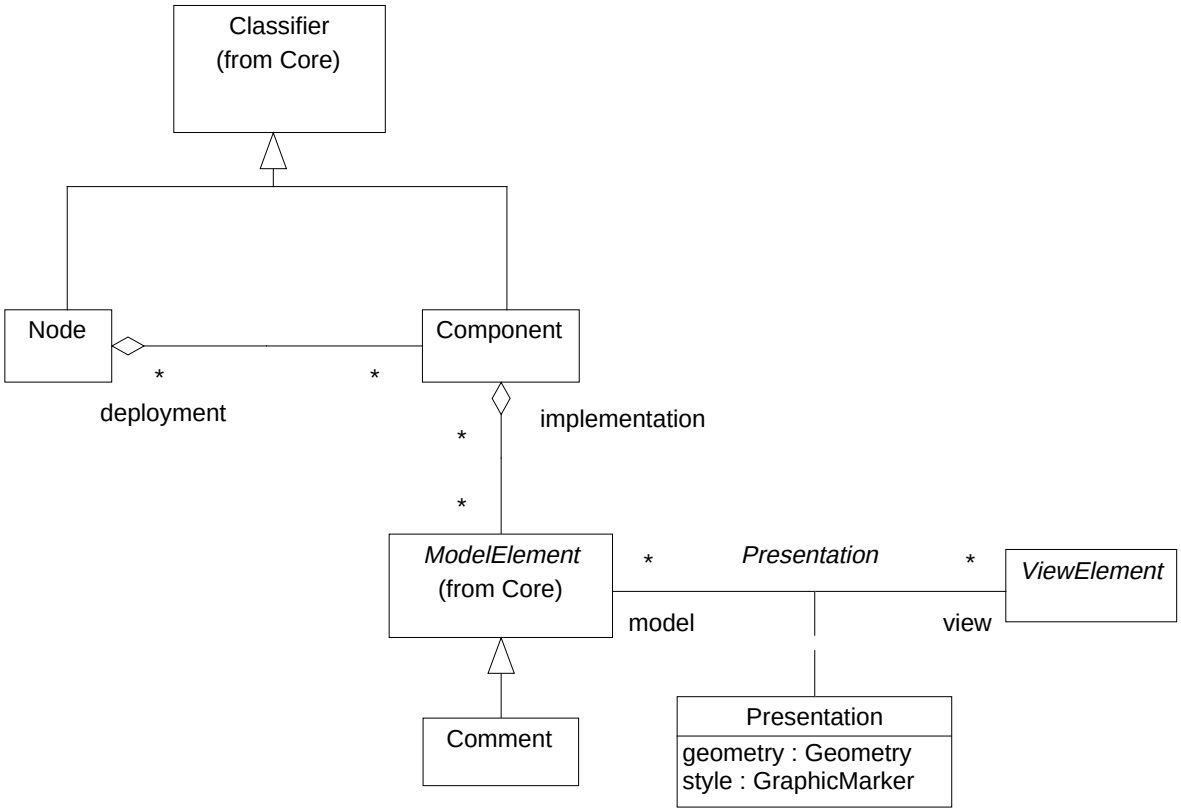


Figure 8: Auxiliary Elements—Physical Structures and View Elements

The following metaclasses are contained in the Auxiliary Elements package:

Binding

A *binding* is a relationship between a template and a model element generated from the template. It includes a list of arguments matching the template parameters. The template is a form that is cloned and modified by substitution to yield an implicit model fragment that behaves as if it were a direct part of the model.

In the metamodel a ***Binding*** is a ***Dependency*** where the supplier is the template and the client is the instantiation of the template that performs the substitution of parameters of a template. A ***Binding*** has a list of arguments that replace the parameters of the supplier to yield the client. The client is fully specified by the binding of the supplier's parameters and does not add any information of its own.

Associations

argument	An ordered list of arguments. Each argument replaces the corresponding supplier parameter in the supplier definition, and the result represents the definition of the client as if it had been defined directly.

Comment

A *comment* is an annotation attached to a model element or a set of model elements.

In the metamodel a ***Comment*** is a subclass of ***ViewElement***. It is associated with a set of

ModelElements.

Component

A *component* is a reusable part that provides the physical packaging of model elements.

In the metamodel a ***Component*** is a subclass of ***Class***. It provides the physical packaging of its associated specification elements.

Associations

deployment	The set of ***Nodes*** the ***Component*** is residing on.

Dependency (from Core)

A *dependency* indicates a semantic relationship among model elements themselves (rather than instances of them) in which a change to one element may affect or require changes to other elements.

In the metamodel a ***Dependency*** is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier, i.e. a change to the supplier may affect the client. The relationship is directed, although the direction may be ignored for certain subtypes of ***Dependency*** (such as ***Trace***).

To enable grouping of dependencies that belong together, a dependency can serve as a container for a group of ***Dependencies***. This is useful, because often dependencies are between groups of elements (such as ***Packages***, ***Models***, ***Classifiers***, etc.). For example, the dependency of one package on another can be expanded into a set of dependencies among elements within the two packages

Associations

client	The element that is affected by the supplier element. In some cases (such as ***Trace***) the direction is unimportant and serves only to distinguish the two elements.
owningDependency	Inverse of subDependency**.**
subDependency	A set of more specific dependencies that elaborate a more general dependency.
supplier	Inverse of **client**. Designates the element that is unaffected by a change. In a two-way relationship (such as some ***Refinements***) this should be the more general element.

ModelElement (from Core)

A *model element* is an element that is an abstraction drawn from the system being modeled. Contrast with *view element*, which is an element whose purpose is to provide a presentation of information for human comprehension.

In the metamodel a ***ModelElement*** is a named entity in a ***Model***. It is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either direct or indirect subclasses of ***ModelElement***.

Each ***ModelElement*** can be regarded as a template. A template has a set of *templateParameters* that denotes which of the parts of a ***ModelElement*** are the template parameters. A ***ModelElement***

is a template when there is at least one template parameter. If it is not a template, a ***ModelElement*** cannot have template parameters. However, such embedded parameters are not usually complete and need not satisfy well-formedness rules. It is the arguments supplied when the template is instantiated that must be well-formed.

Partially instantiated templates are allowed. This is the case when there are arguments provided for some, but not all templateParameters. A partially instantiated template is still a template, since it has still parameters.

Associations

templateParameter An ordered list of parameters. Each parameter designates a ***ModelElement*** within the scope of the overall ***ModelElement***. The designated ***ModelElement*** may be a placeholder for a real ***ModelElement*** to be substituted. In particular, the template parameter element will lack structure. For example, a parameter that is a ***Class*** lacks ***Features***; they are found in the actual argument.

Node

A *node* is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed.

In the metamodel a ***Node*** is a subclass of ***Class***. It is associated with a set of ***Components*** residing on the ***Node***.

Associations

component The set of ***Components*** residing on the ***Node***.

Presentation

A *presentation* is the relationship between a view element and a model element (or possibly a set of each). The details are dependent on the implementation of a graphic editor tool.

In the metamodel ***Presentation*** reifies the relationship between ***ModelElement*** and ***ViewElement*** and provides the placement and the style of presentation to be used when presenting the ***ModelElements***.

Attributes

geometry A description of the geometry of the ***ViewElement*** image.

style A description of the graphic markers pertaining to the ***ViewElement*** image, such as color, texture, font, line width, shading, etc.

Refinement

A *refinement* is a relationship between model elements at different semantics levels, such as analysis and design.

In the metamodel a ***Refinement*** is a ***Dependency*** where the clients are derived from the suppliers. The derivation cannot necessarily be described by an algorithm; human decisions may be required to produce the clients. The details of specifying the derivation are beyond the scope of UML but

can be indicated with constraints. ***Refinement*** can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

Associations

mapping	A description of the mapping between the two elements. The mapping is an expression whose syntax is beyond the scope of UML. For exchange purposes it should be represented as a string.

Trace

A *trace* is a conceptual connection between two elements or sets of elements that represent a single concept at different semantic levels or from different points of view. However, there is no specific mapping between the elements. The construct is mainly a tool for tracing of requirements. It is also useful for the modeler to keep track of changes to different models.

In the metamodel a ***Trace*** is a ***Dependency*** between ***ModelElements*** in different ***Models*** abstracting the same part of the system being modeled. ***Traces*** therefore denote dependencies at specification level, rather than runtime dependencies. Thus, ***Traces*** do not express information on the system as such, but rather on the ***Models*** of the system. The directionality of the dependency can usually be ignored.

Usage

A *usage* is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. The relationship is not a mere historical artifact but an ongoing need. Therefore two elements related by *usage* must be in the same model.

In the metamodel a ***Usage*** is a ***Dependency*** in which the client requires the presence of the supplier. How the client uses the supplier, such as a class calling an operation of another class, a method having an argument of another class, and a method from a class instantiating another class, is defined in the description of the ***Usage***.

ViewElement

A *view element* is a textual or graphical presentation of one or more model elements.

In the metamodel a ***ViewElement*** is an ***Element*** which presents a set of ***ModelElements*** to a reader. It is the base for all metaclasses in the UML used for presentation. All other metaclasses with this purpose are either direct or indirect subclasses of ***ViewElement***. ***ViewElement*** is an abstract metaclass. The subclasses of this class are proper to a graphic editor tool and are not specified here.

5.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Auxiliary Elements package.

Binding

[1] The *argument* ***ModelElement*** must conform to the *parameter* ***ModelElement*** in a Binding. In an instantiation it must be of the same kind.

```
-- not described in OCL
```

Comment

No extra well-formedness rules.

Component

No extra well-formedness rules.

Dependency

No extra well-formedness rules.

Additional operations

[1] A ***Dependency*** is a composite dependency if it contains other dependencies.

```
isComposite : Boolean;
isComposite = (self.subDependency->size >= 1);
```

ModelElement

A model element *owns* everything connected to it by composition relationships.

A *template* is a model element with at least one template parameter.

That part of the model owned by a template is not subject to all well-formedness rules. A template is not directly usable in a well-formed model. The results of binding a template are subject to well-formedness rules.

Additional operations

[1] A ***ModelElement*** is a template when is has parameters.

```
isTemplate : Boolean;
isTemplate = (self.templateParameter->notEmpty)
```

[2] A ***ModelElement*** is an instantiated template when it related to a template by a ***Binding*** relationship.

```
isInstantiated : Boolean;
isInstantiated = self.requirement->select(oclIsKindOf(Binding))->notEmpty
```

[3] The *templateArguments* are the arguments of an instantiated template, which substitute for template parameters.

```
templateArguments : Set(ModelElement);
templateArguments = self.requirement->
         select(oclIsKindOf(Binding)).oclAsType(Binding).argument
```

Node

No extra well-formedness rules.

Presentation

No extra well-formedness rules.

Refinement

No extra well-formedness rules.

Trace

[1] A ***Trace*** connects two sets of ***ModelElements*** from two different ***Models*** in the same ***System***.

```
self.client->forAll( e1, e2 | e1.model = e2.model ) and
self.supplier->forAll( e1, e2 | e1.model = e2.model ) and
self.client->asSequence->at (1).model <>
      self.supplier->asSequence->at (1).model and
self.client->asSequence->at (1).model.namespace =
      self.supplier->asSequence->at (1).model.namespace
```

Usage

No extra well-formedness rules.

ViewElement

No extra well-formedness rules.

5.4 SEMANTICS

Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possibly changes to the derived client element. Such a check should be performed after which action can be taken to change the derived element to validate it again. The semantics of this validation and change is outside the scope of UML.

Template

An important dynamic consequence is that any model element that is a template cannot be instantiated. Only a fully instantiated model element can have instances. This applies specifically to classifier templates.

Also a template is a *form*, not a final model element. As such, it is not subject to normal well-formedness rules because it is intentionally incomplete. Only when a template is bound with arguments can the result be fully subject to well-formedness rules.

A further consequence is that a template must *own* a fragment of the model that is not part of the final effective model. When a template is bound, the model fragment that it owns is implicitly duplicated, the parameters are replaced by the arguments, and the result is implicitly added to the effective model, *as if* the effective model had been modeled directly.

ViewElement

The responsibility of *view element* is to provide a textual and graphical projection of a collection of model elements. In this context projection means that the view element represents a human readable notation for the corresponding model elements. The notation for UML can be found in a separate document.

View elements and model elements must be kept in agreement, but the mechanisms for doing this are design issues for model editing tools.

5.5 STANDARD ELEMENTS

The predefined stereotypes, constraints and tagged values for the Auxiliary Elements package are

listed in Table 3 and defined in the *Standard Elements* appendix.

Table 3: Auxiliary Elements—Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Comment	«requirement»		location
Component	«document» «executable» «file» «friend» «library» «table»		
Dependency	«becomes» «call» «copy» «deletion» «derived» «import» «instance» «metaclass» «powertype» «send»		

6. FOUNDATION PACKAGE: EXTENSION MECHANISMS

6.1 OVERVIEW

The Extension Mechanisms package is the subpackage of the Foundation package that specifies how model elements are customized and extended with new semantics. It defines the semantics for stereotypes, constraints, and tagged values.

The UML provides a rich set of modeling concepts and notations that have been carefully designed to meet the needs of typical software modeling projects. However, users may sometimes require additional features and/or notations beyond those defined in the UML standard. In addition, users often need to attach non-semantic information to models. These needs are met in UML by three built-in extension mechanisms that enable new kinds of modeling elements to be added to the modeler's repertoire as well as to attach free-form information to modeling elements. These three extension mechanisms can be used separately or together to define new modeling elements that can have distinct semantics, characteristics and notation relative to the built in UML modeling elements specified by the UML metamodel. Concrete constructs defined in Extension Mechanisms include ***Constraint***, ***Stereotype,*** and ***TaggedValue.***

The UML extension mechanisms are intended for several purposes:

- They can be used to add new modeling elements for use in creating UML models.
- They are used in the UML specification to define standard items that are not considered interesting or complex enough to be defined directly as UML metamodel elements.
- They are used to define process-specific or implementation language-specific extensions.
- They are used to attach arbitrary semantic and non-semantic information to model elements.

Although it is beyond the scope and intent of this document, it is also possible to extend the UML metamodel by explicitly adding new metaclasses and other meta constructs. This capability depends on unique features of certain UML-compatible modeling tools, or direct use of a meta-metamodel facility, such as the CORBA Meta Object Facility.

The most important of the built-in extension mechanisms is based on the concept of *Stereotype*. Stereotypes provide a way of classifying model elements at the object model level and facilitate the addition of "virtual" UML metaclasses with new metaattributes and semantics. The other built in extension mechanisms are based on the notion of property lists consisting of *tags* and *values*, and *constraints.* These allow users to attach additional properties and semantics directly to individual model elements, as well as to model elements classified by a Stereotype.

A stereotype is a UML model element that is used to classify (or mark) other UML elements so that they behave in some respects as if they were instances of new "virtual" or "pseudo" metamodel classes whose form is based on existing "base" classes. Stereotypes thus augment the classification mechanism based on the built in UML metamodel class hierarchy, and therefore names of new stereotypes must not clash with the names of predefined metamodel elements or other stereotypes. Any model element can be marked by at most one stereotype, but any stereotype can be constructed as a specialization of numerous other stereotypes.

A stereotype may introduce additional values, additional constraints and a new graphical representation. All model elements that are classified by a particular stereotype ("stereotyped") receive these values, constraints and representation. By allowing stereotypes to have associated graphical representations users can introduce new ways of graphically distinguishing model elements classified by a particular stereotype.

A stereotype shares the attributes, associations, and operations of its base class but it may have additional well-formedness constraints as well as a different meaning and attached values. The intent is that a tool or repository be able to manipulate a stereotyped element the same as the same ordinary element for most editing and storage purposes, while differentiating it for certain semantic operations, such as well-formedness checking, code generation, or report writing.

Any modeling element may have arbitrary attached information in the form of a property list consisting of tag-value pairs. A tag is a name string that is unique for a given element that selects an associated arbitrary value. Values may be arbitrary but for uniform information exchange they should be represented as strings. The tag represents the name of an arbitrary property with the given value. Tags may be used to represent management information (*author, due date, status*), code generation information (*optimizationLevel*, *containerClass*), or additional semantic information required by a given stereotype

It is possible to specify a list of tags (with default values, if desired) that are required by a particular stereotype. Such required tags serve as "pseudoattributes" of the stereotype to supplement the real attributes supplied by the base element class. The values permitted to such tags can also be constrained.

It is not necessary to stereotype a model element in order to give it individually distinct constraints or tagged values. Constraints can be directly attached to a model element (stereotyped or not) to change its semantics. Likewise, a property list consisting of tag-value pairs can be directly attached to any model element. The tagged values of a property list allow characteristics to be assigned to model elements on a flexible, individual basis. Tags are user-definable; certain ones are predefined and are listed in the *Standard Elements* appendix.

Constraints or tagged values associated with a particular stereotype are used to extend the semantics of model elements classified by that stereotype. The constraints must be observed by all model elements marked with that stereotype.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Extension Mechanisms package.

6.2 ABSTRACT SYNTAX

The abstract syntax for the Extension Mechanisms package is expressed in graphic notation in Figure 9.

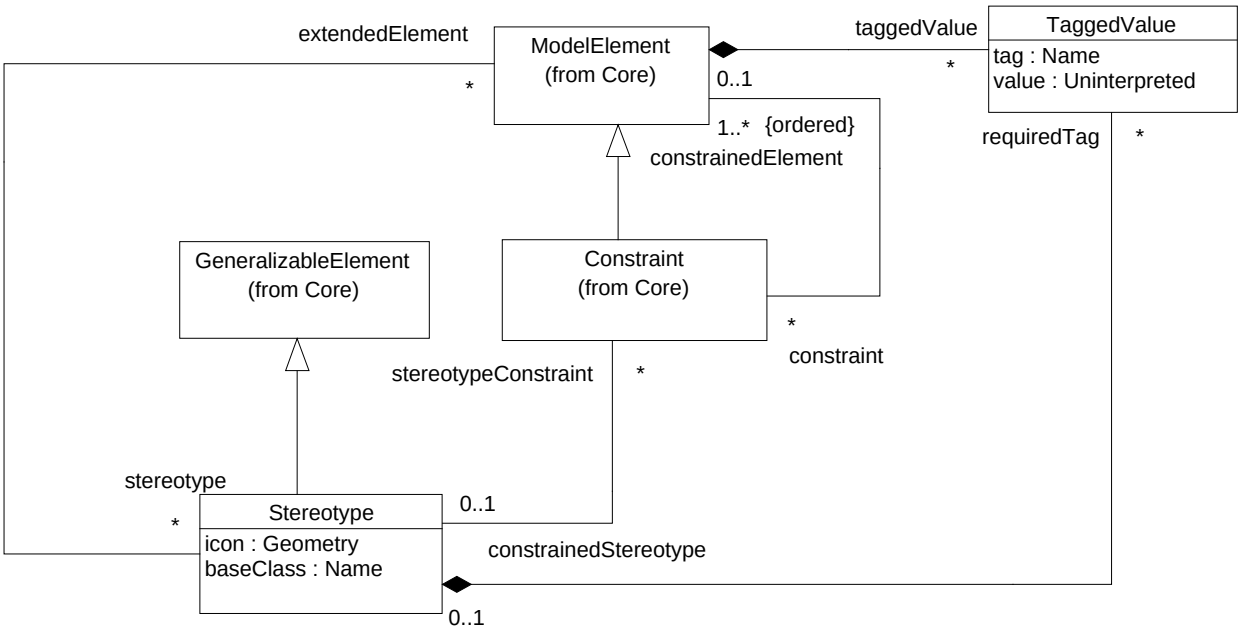


Figure 9: Extension Mechanisms

The following metaclasses are contained in the Extension Mechanisms package:

Constraint

The *constraint* concept allows new semantics to be specified linguistically for a model element. The specification is written as an expression in a designated constraint language. The language can be specially designed for writing constraints (such as OCL), a programming language, mathematical notation, or natural language. If constraints are to be enforced by a model editor tool, then the tool must understand the syntax and semantics of the constraint language. Because the choice of language is arbitrary, constraints are an extension mechanism.

In the metamodel, a ***Constraint*** directly attached to a ***ModelElement*** describes semantic restrictions that this ***ModelElement*** must obey. Also, any ***Constraints*** attached to a ***Stereotype*** apply to each ***ModelElement*** that bears the given ***Stereotype***.

Attributes

body	A boolean expression defining the constraint. Expressions are written as strings in a designated language. For the model to be well formed, the expression must always yield a true value when evaluated for instances of the constrained elements at any time when the system is stable (i.e., not during the execution of an atomic operation).

Associations

constrainedElement An ordered list of elements subject to the constraint. The constraint applies to their instances.

constrainedStereotype An ordered list of stereotypes subject to the constraint. The constraint applies to instances of elements classified by the stereotypes.

Any particular constraint has either a ***constrainedElement*** link or a ***constrainedStereotype*** link but not both.

ModelElement (as extended)

Any model element may have arbitrary tagged values and constraints (subject to theses making sense). A model element may have at most one stereotype whose base class must match the UML class of the modeling element (such as Class, Association, Dependency, etc.). The presence of a stereotype may impose implicit constraints on the modeling element and may require the presence of specific tagged values.

Associations

constraint A constraint that must be satisfied for instances of the model element. A model element may have a set of constraints. The constraint is to be evaluated when the system is stable, i.e., not in the middle of an atomic operation.

stereotype Designates at most one stereotype that further qualifies the UML class (the *base class*) of the modeling element. The stereotype does not alter the structure of the base class but it may specify additional constraints and tagged values. All constraints and tagged values on a stereotype apply to the model elements that are classified by the stereotype. The stereotype acts as a "pseudo metaclass" describing the model element.

taggedValue An arbitrary property attached to the model element. The tag is the name of the property and the value is an arbitrary value. The interpretation of the tagged value is outside the scope of the UML metamodel. A model element may have a set of tagged values, but a single model element may have at most one tagged value with a given tag name. If the model element has a stereotype, then it may specify that certain tags must be present, providing default values.

Stereotype

The *stereotype* concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs. Instances have the same structure (attributes, associations, operations) as a similar non-stereotyped instance of the same kind; the stereotype may specify additional constraints and required tagged values that apply to instances. In addition, a stereotype may be used to indicate a difference in meaning or usage between two elements with identical structure.

In the metamodel, the ***Stereotype*** metaclass is a subtype of ***GeneralizableElement***. ***TaggedValues*** and ***Constraints*** attached to a ***Stereotype*** apply to all ***ModelElements*** classified by that ***Stereotype***. A stereotype may also specify a geometrical icon to be used for presenting elements with the stereotype

Stereotypes are ***GeneralizableElements***. If a stereotype is a subtype of another stereotype, then it

inherits all of the constraints and tagged values from its stereotype supertype and it must apply to the same kind of base class. . A stereotype keeps track of the base class to which it may be applied.

Attributes

baseClass Species the name of a UML modeling element to which the stereotype applies, such as ***Class, Association, Refinement, Constraint***, etc. This is the name of a metaclass, that is, a class from the UML metamodel itself rather than a user model class.

icon The geometrical description for an icon to be used to present an image of a model element classified by the stereotype.

Associations

extendedElement Designates the model elements affected by the stereotype. Each one must be a model element of the kind specified by the *baseClass* attribute.

stereotypeConstraint Designates constraints that apply to elements bearing the stereotype.

requiredTag Specifies a set of tagged values, each of which specifies a tag that an element classified by the stereotype is required to have. The value part indicates the default value for the tag-value, that is, the tag-value that an element will be presumed to have if it is not overridden by an explicit tagged value on the element bearing the stereotype. If the value is *unspecified* then the element must explicitly specify a tagged value with the given tag.

TaggedValue

A tagged value is a (Tag, Value) pair that permits arbitrary information to be attached to any model element. A tag is an arbitrary name; some tag names are predefined as Standard Elements (see). At most one tagged value pair with a given tag name may be attached to a given model element. In other words, there is a lookup table of values selected by tag strings that may be attached to any model element.

The interpretation of a tag is (intentionally) beyond the scope of UML; it must be determined by user or tool convention. It is expected that various model analysis tools will define tags to supply information needed for their operation beyond the basic semantics of UML. Such information could include code generation options, model management information, or user-specified additional semantics.

Attributes

tag A name that indicates an extensible property to be attached to ***ModelElements***. There is a single, flat space of tag names. UML does not define a mechanism for name registry but model editing tools are expected to provide this kind of service. A model element may have at most one tagged value with a given name. A tag is, in effect, a pseudoattribute that may be attached to model elements.

value An arbitrary value. The value must be expressible as a string for uniform manipulation. The range of permissible values depends on the interpretation applied to the tag by the user or tool; its specification is outside the scope of UML.

Associations

taggedValue — A ***TaggedValue*** that is attached to a ***ModelElement***.

requiredTag — A ***TaggedValue*** that is attached to a ***Stereotype***. A particular ***TaggedValue*** can be attached to either a ***ModelElement*** or a ***Stereotype***, but not both.

6.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Extension Mechanisms package.

Constraint

[1] A *Constraint* attached to a *Stereotype* must not conflict with Constraints on any inherited Stereotype, or associated with the *baseClass*.

```
-- cannot be specified with OCL
```

[2] A Constraint attached to a stereotyped ModelElement must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the *baseClass*) of the ModelElement

```
-- cannot be specified with OCL
```

[3] A Constraint attached to a Stereotype will apply to all ModelElements classified by that Stereotype and must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the *aseClass*) of the ModelElement

```
-- cannot be specified with OCL
```

Stereotype

[1] *Stereotype* names must not clash with any *baseClass* names

```
Stereotype.oclAllInstances->forAll(st | st.baseClass <> self.name)
```

[2] *Stereotype* names must not clash with the names of any inherited Stereotype.

```
self.allSupertypes->forAll(st : Stereotype | st.name <> self.name)
```

[3] *Stereotype* names must not clash in the (M2) meta-class namespace, nor with the names of any inherited Stereotype, nor with any *baseClass* names

```
-- M2 level not accessible
```

[4] The *baseClass* name must be provided; *icon* is optional is specified in an implementation specific way

```
self.baseClass <> ''
```

[5] Tag names attached to a Stereotype must not clash with M2 meta-attribute namespace of the appropriate *baseClass* element, nor with Tag names of any inherited Stereotype

```
-- M2 level not accessible
```

ModelElement

[1] Tags associated with a ModelElement (directly via a property list or indirectly via a Stereotype) must not clash with any meta-attributes associated with the Model Element.

```
-- not specified in OCL
```

[2] A model element must have at most one tagged value with a given tag name.

```
self.taggedValue->forAll(t1, t2 : TaggedValue |
   t1.tag = t2.tag implies t1 = t2)
```

[3] (Required tags because of stereotypes) If T in modelElement.stereotype.requiredTag.such that T.value = **unspecified** then the modelElement must have a tagged value with name = T.name.

```
self.stereotype.requiredTag->forAll(tag |
      tag.value = Undefined implies self.taggedValue->exists(t |
            t.tag = tag.tag))
```

TaggedValue

No extra well-formedness rules.

6.4 SEMANTICS

Constraints, stereotypes, and tagged values apply to model elements, not to instances. They represent extensions to the modeling language itself, not extensions to the run-time environment. They affect the structure and semantics of models. These concepts represent metalevel extensions to UML; however, they do not contain the full power of a heavyweight metamodel extension language and they are designed such that tools need not implement metalevel semantics to implement them.

Within a model, any user-level model element may have a set of constraints and a set of tagged values. The constraints specify restrictions on the instantiation of the model. An instance of a user-level model element must satisfy all of the constraints on its model element for the model to be well-formed. Evaluation of constraints is to be performed when the system is “stable”, that is, after the completion of any internal operations when it is waiting for external events. Constraints are written in a designated constraint language, such as OCL, C++, or natural language. The interpretation of the constraints must be specified by the constraint language.

A user-level model element may have at most one tagged value with a given tag name. Each tag name represents a user-defined property applicable to model elements with a unique value for any single model element. The meaning of a tag is outside the scope of UML and must be determined by convention among users and model analysis tools.

It is intended that both constraints and tagged values be represented as strings so that they can be edited, stored, and transferred by tools that may not understand their semantics. The idea is that the understanding of the semantics can be localized into a few modules that make use of the values. For example, a code generator could use tagged values to tailor the code generation process and a process planning tool could use tagged values to denote model element ownership and status. Other modules would simply preserve the uninterpreted values (as strings) unchanged.

A stereotype refers to a *baseClass*, which is a class in the UML metamodel (not a user-level modeling element) such as ***Class, Association, Refinement***, etc. A stereotype may be a subtype of one or more existing stereotypes (which must all refer the same baseClass, or baseClasses that derive from the same baseClass), in which case it inherits their constraints and required tags and may add additional ones of its own. As appropriate, a stereotype may add new constraints, a new icon for visual display, and a list of default tagged values.

If a user-level model element is classified by an attached stereotype, then the UML base class of the model element must match the base class specified by the stereotype. Any constraints on the

stereotype are implicitly attached to the model element. Any tagged values on the stereotype are implicitly attached to the model element; if any of the values are ***unspecified***, then the model element must explicitly define tagged values with the same tag name or the model is ill-formed. (This behaves as if a copy of the tagged values from the stereotype is attached to the model element, so that the default values can be changed). If the stereotype is a subtype of one or more other stereotypes, then any constraints or tagged values from those stereotypes also apply to the model element (because they are inherited by this stereotype). If there are any conflicts among multiple constraints or tagged values (inherited or directly specified) then the model is ill-formed.

6.5 STANDARD ELEMENTS

None.

6.6 NOTES

From an implementation point of view, instances of a stereotyped class are stored as instances of the base class with the stereotype name as a property. Tagged values can and should be implemented as a lookup table (qualified association) of values (expressed as strings) selected by tag names (represented as strings). Attributes of UML metamodel classes and tag names should be accessible using a single uniform string-based selection mechanism. This allows tags to be treated as pseudo-attributes of the metamodel and stereotypes to be treated as pseudo-classes of the metamodel, permitting a smooth transition to a full metamodeling capability if desired. See the *UML Proposal Summary* for discussion of how the OMG Meta Object Facility can be used for extending the metamodel.

7. FOUNDATION PACKAGE: DATA TYPES

7.1 OVERVIEW

The Data Types package is the subpackage of the Foundation package that specifies the different data types used by UML. This section has a simpler structure than the other packages, since it is assumed that the semantics of these basic concepts are well known.

The following sections describes the abstract syntax of the Data Types package

7.2 ABSTRACT SYNTAX

The abstract syntax for the Data Types package is expressed in graphic notation in Figure 10.

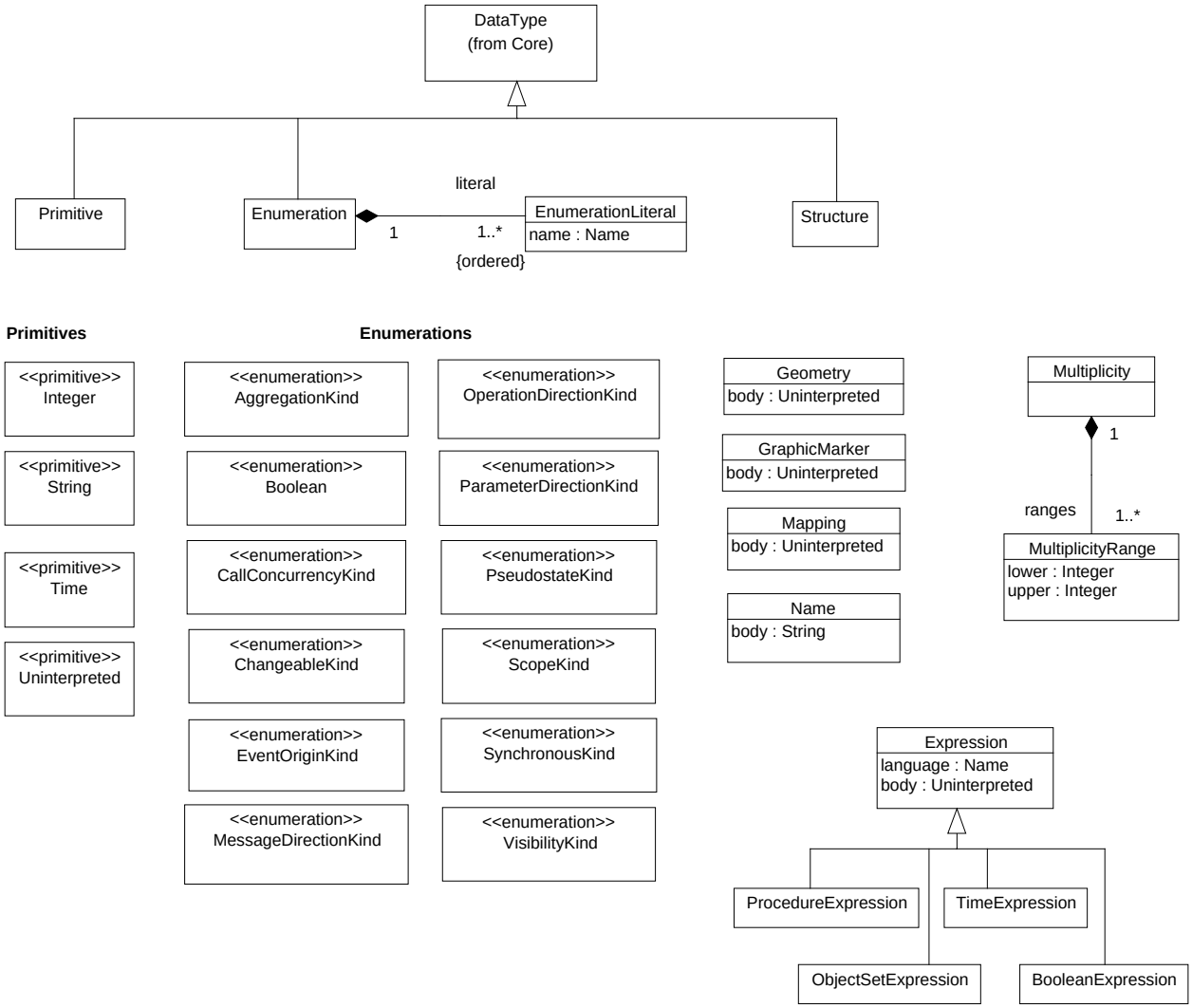


Figure 10: Data Types

In the metamodel the data types are used for declaring the types of the classes' attributes. They appear as strings in the diagrams and not with a separate 'data type' icon. In this way, the sizes of the diagrams are reduced. However, each occurrence of a particular name of a data type denotes the same data type.

Note that these data types are the data types used for defining UML and not the data types to be used by a user of UML. The latter data types will be instances of the ***DataType*** metaclass defined in the metamodel.

AggregationKind

In the metamodel ***AggregationKind*** defines an enumeration whose values are *none*, *shared*, and

composite. Its value denotes what kind of aggregation an ***Association*** is.

Boolean

In the metamodel ***Boolean*** defines an enumeration whose values are *false* and *true*.

BooleanExpression

In the metamodel ***BooleanExpression*** defines a statement which will evaluate to an instance of ***Boolean*** when it is evaluated.

ChangeableKind

In the metamodel ***ChangeableKind*** defines an enumeration whose values are *none*, *frozen*, and *addOnly*. Its value denotes how an ***AttributeLink*** or ***LinkEnd*** may be modified.

Enumeration

In the metamodel ***Enumeration*** defines a special kind of ***DataType*** whose range is a list of definable values, called ***EnumerationLiterals***.

EnumerationLiteral

An ***EnumerationLiteral*** defines an atom (i.e., with no relevant substructure) that but can be compared for equality.

Expression

In the metamodel an ***Expression*** defines a statement which will evaluate to a (possibly empty) set of instances when executed in a context. An ***Expression*** does not modify the environment in which it is evaluated.

Geometry

In the metamodel a ***Geometry*** is a triple of ***Floats***, which denotes a position in space.

GraphicMarker

In the metamodel ***GraphicMarker*** defines the presentation characteristics of view elements, such as color, texture, font, line width, shading, etc

Integer

In the metamodel an ***Integer*** is an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…).

Mapping

In the metamodel a ***Mapping*** is an expression that is used for mapping ***ModelElements***. For exchange purposes it should be represented as a ***String***.

MessageDirectionKind

In the metamodel ***MessageDirectionKind*** defines an enumeration whose values are *activation* and *return*. Its value denotes the direction of a ***Message***.

Multiplicity

In the metamodel a ***Multiplicity*** defines a non-empty set of non-negative integers. A set which only

contains zero ({0}) is not considered a valid ***Multiplicity***. Every ***Multiplicity*** has at least one corresponding ***String*** representation.

MultiplicityRange

In the metamodel a ***MultiplicityRange*** defines a range of integers. The *upper* bound of the range cannot be below the *lower* bound.

Name

In the metamodel a ***Name*** defines a token which is used for naming ***ModelElements***. Each ***Name*** has a corresponding ***String*** representation.

ObjectSetExpression

In the metamodel ***ObjectSetExpression*** defines a statement which will evaluate to a set of instances when it is evaluated. ***ObjectSetExpressions*** are commonly used to designate the target instances in an ***Action***.

OperationDirectionKind

In the metamodel ***OperationDirectionKind*** defines an enumeration whose values are *provide* and *require*. Its value denotes if an ***Operation*** is required or provided by a ***Classifier***.

ParameterDirectionKind

In the metamodel ***ParameterDirectionKind*** defines an enumeration whose values are *in*, *inout*, *out*, and *return*. Its value denotes if a ***Parameter*** is used for supplying an argument and/or for returning a value.

Primitive

A ***Primitive*** defines a special kind of simple ***DataType***, without any relevant substructure.

ProcedureExpression

In the metamodel ***ProcedureExpression*** defines a statement which will result in an instance of ***Procedure*** when it is evaluated.

PseudostateKind

In the metamodel ***VisibilityKind*** defines an enumeration whose values are *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *branch*, and *final*. Its value denotes the possible pseudo states in a state machine.

ScopeKind

In the metamodel ***ScopeKind*** defines an enumeration whose values are *classifier* and *instance*. Its value denotes if the stored value should be an instance of the associated ***Classifier*** or the ***Classifier*** itself.

String

In the metamodel a ***Sting*** defines a stream of text.

Structure

A ***Structure*** defines a special kind of ***DataType***, that has a fixed number of named parts.

SynchronousKind

In the metamodel ***SynchronousKind*** defines an enumeration whose values are *synchronous* and *asynchronous*. Its value denotes what kind of ***Message*** a ***CallAction*** will create when executed.

Time

In the metamodel a ***Time*** defines a value representing an absolute or relative moment in time and space. A ***Time*** has a corresponding string representation.

TimeExpression

In the metamodel ***TimeExpression*** defines a statement which will evaluate to an instance of ***Time*** when it is evaluated.

Uninterpreted

In the metamodel an ***Uninterpreted*** is a blob, the meaning of which is domain-specific and therefore not defined in UML.

VisibilityKind

In the metamodel ***VisibilityKind*** defines an enumeration whose values are *public*, *protected*, and *private*. Its value denotes how the element to which it refers is seen outside the enclosing name space.

PART 3. BEHAVIORAL ELEMENTS

Part 3 defines the superstructure for behavioral modeling in UML, the Behavioral Elements package. The Behavioral Elements package consists of four lower-level packages: Common Behavior, Collaborations, Use Cases, and State Machines. Common Behavior specifies the core concepts required for behavioral elements. The Collaborations package specifies a behavioral context for using model elements to accomplish a particular task. The Use Case package specifies behavior using actors and use cases. The State Machines package defines behavior using finite-state transition systems.

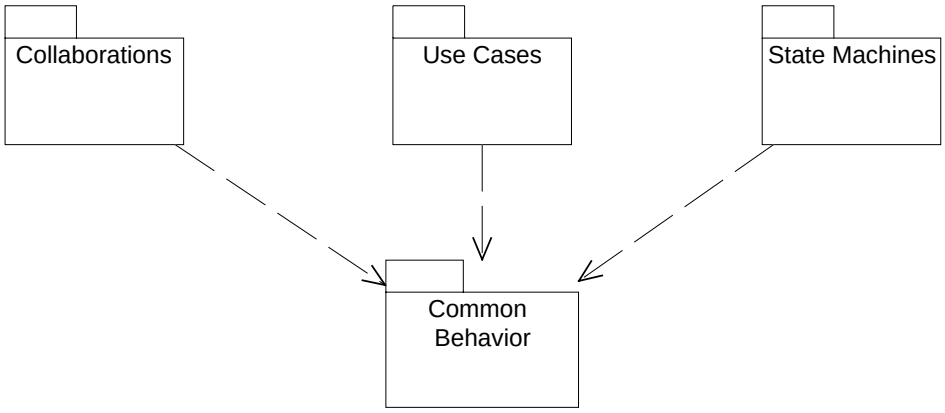


Figure 11: Behavioral Elements Packages

Contents

8. Behavioral Elements Package: Common Behavior

9. Behavioral Elements Package: Collaborations

10. Behavioral Elements Package: Use Cases

11. Behavioral Elements Package: State Machines

8. BEHAVIORAL ELEMENTS PACKAGE: COMMON BEHAVIOR

8.1 OVERVIEW

The Common Behavior package is the most fundamental of the subpackages that compose the Behavioral Elements package. It specifies the core concepts required for dynamic elements and provides the infrastructure to support Collaborations, State Machines and Use Cases.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Common Behavior package.

8.2 ABSTRACT SYNTAX

The abstract syntax for the Common Behavior package is expressed in graphic notation in Figure 12, Figure 13 and Figure 14. Figure 12 shows the model elements that define ***Requests***, which include ***Signals*** and ***Operations***. Figure 13 illustrates the model elements that specify various actions, such as ***CreateAction***, ***CallAction*** and ***SendAction***. Figure 14 shows the model elements that define ***Instances*** and ***Links***.

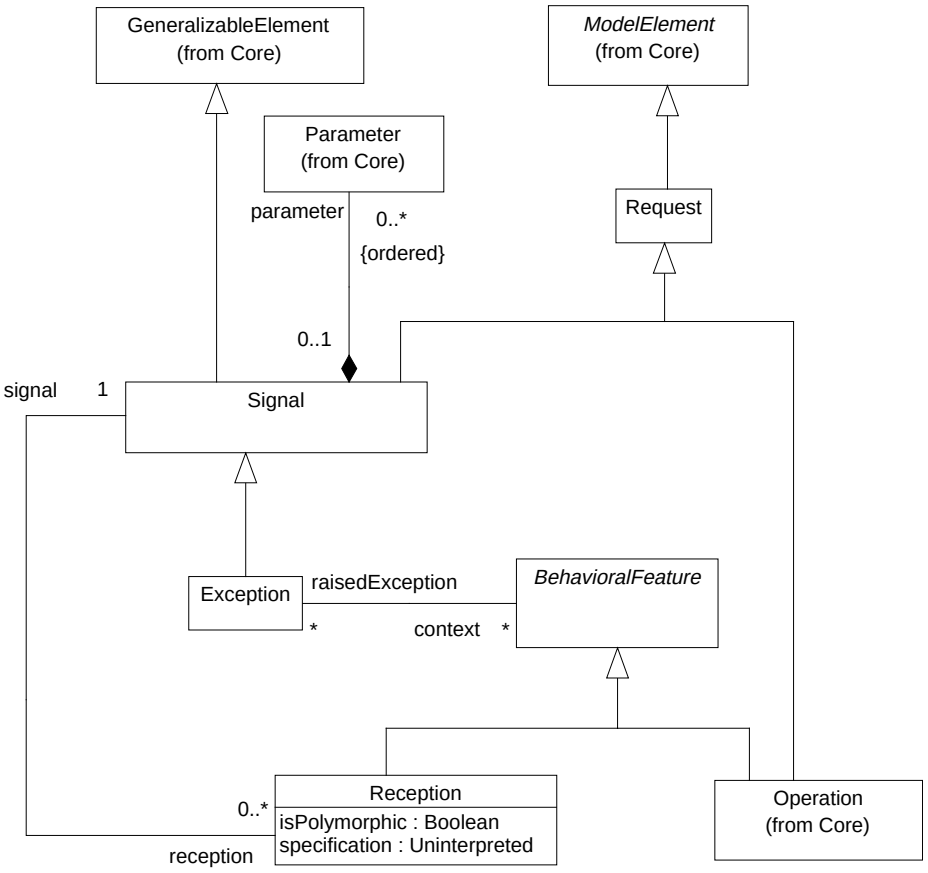


Figure 12: Common Behavior—Requests

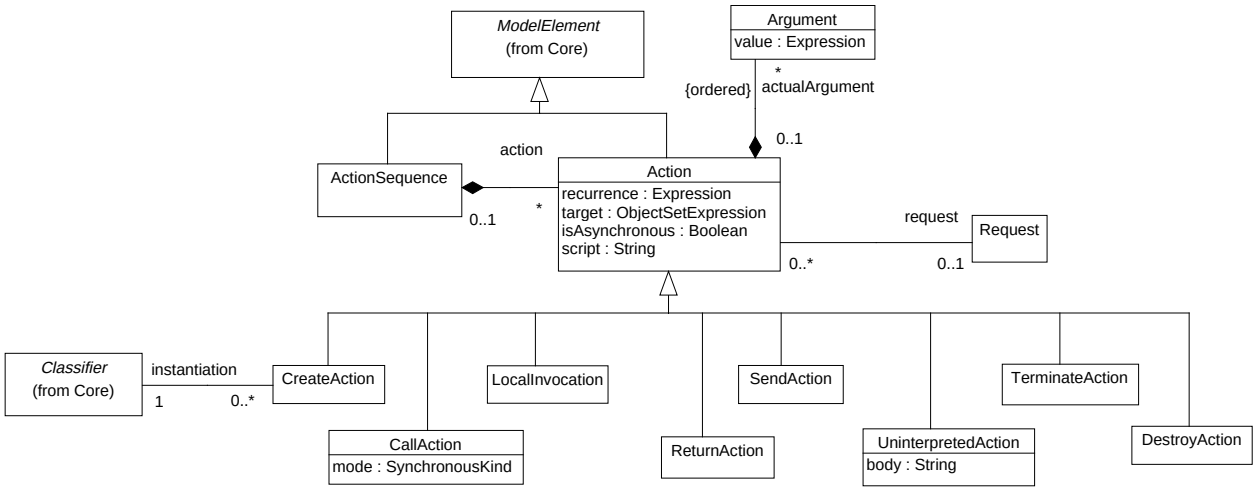


Figure 13: Common Behavior—Actions

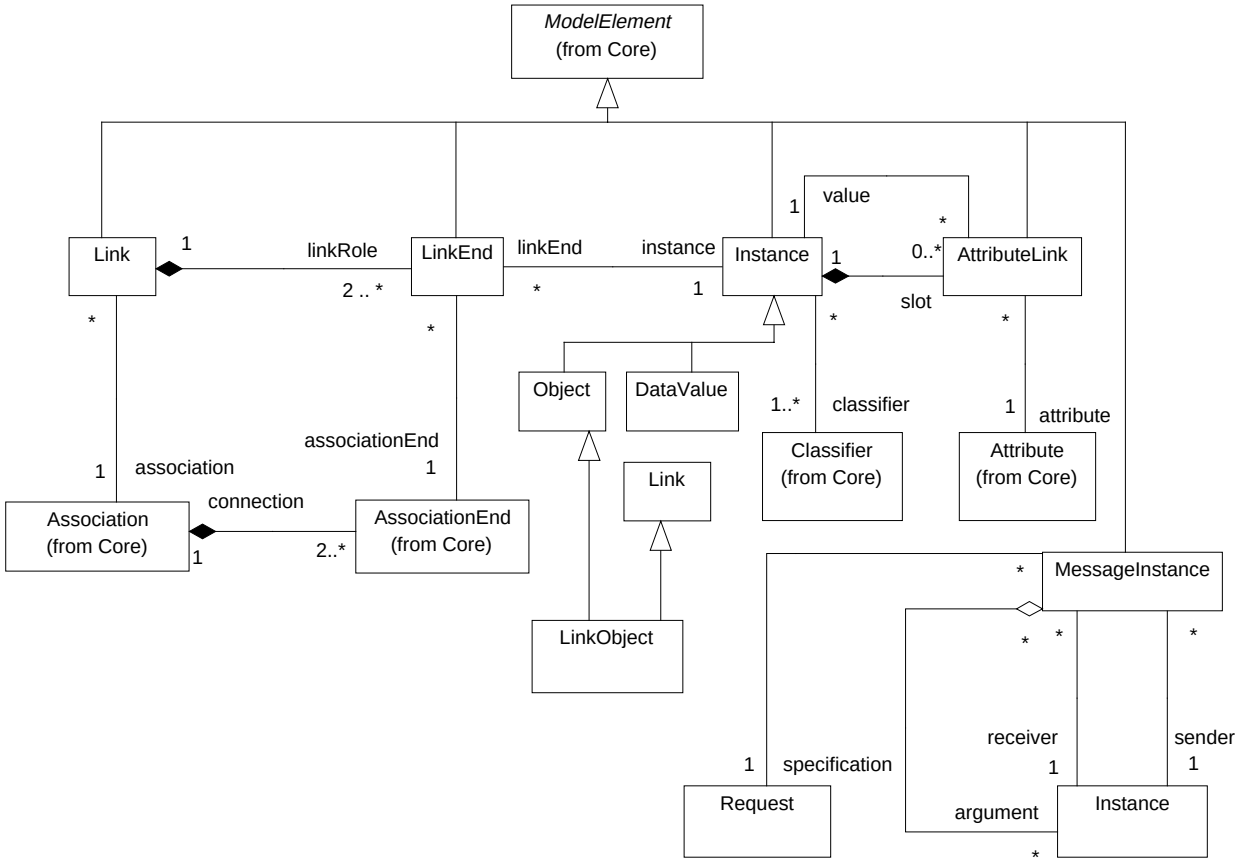


Figure 14: Common Behavior—Instances and Links

The following metaclasses are contained in the Common Behavior package:

Action

An *action* is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, realized by sending a message to an object or modifying a value of an attribute.

In the metamodel an ***Action*** is a part of an ***ActionSequence*** and may contain a specification of a *target* as well as a specification of the *arguments* (actual parameters) of the dispatched ***Request***.

The *target* metaattribute is of type ***ObjectSetExpression*** which, when executed, resolves into zero or more specific ***Instances*** which are the intended recipients of the dispatched ***Request***. Similarly, it is associated with a list of ***Arguments*** which at runtime are resolved to the actual arguments of the ***Request***. The *recurrence* metaattribute specifies how many times the resulted ***Request*** should be sent every time the ***Action*** is executed.

Action is an abstract metaclass.

Attributes

recurrence An ***Expression*** stating how many times the ***Action*** should be performed.

target An ***ObjectSetExpression*** which determines the target of the ***Action***.

Associations

request The specification of the ***Request*** being dispatched by the ***Action***.

actualArgument A sequence of ***Expressions*** which determines the actual arguments needed when evaluating the ***Action***.

ActionSequence

An *action sequence* is a collection of actions.

In the metamodel an ***ActionSequence*** is an aggregation of ***Actions.*** It describes the behavior of the owning ***State*** or ***Transition***.

Associations

action A sequence of ***Actions*** performed sequentially as an atomic unit.

Argument

An *argument* represents the actual values passed to a dispatched request and aggregated within an action.

In the metamodel, an ***Argument*** is a part of an ***Action*** and contains a metaattribute, *value*, of type ***Expression***.

Attributes

value An ***Expression*** determining the actual ***Instance*** when evaluated.

AttributeLink

An *attribute link* is a named slot in an instance, which holds the value of an attribute.

In the metamodel ***AttributeLink*** is a piece of the state of an ***Instance*** and holds the value of an ***Attribute***.

Associations

value — The ***Instance*** which is the value of the ***AttributeLink***.

attribute — The ***Attribute*** from which the ***AttributeLink*** originates.

CallAction

A *call action* is an action resulting in an invocation of an operation on an instance. A call action can be synchronous or asynchronous, indicating whether the operation is invoked synchronously or asynchronously.

In the metamodel, the ***CallAction*** is a subtype of ***Action.*** The designated instance or set of instances is specified via the *target* expression, and the actual arguments are designated via the *argument* association inherited from ***Action.*** The resulting operation is specified by the dispatched ***Request***, which in that case should be an ***Operation***.

Attributes

mode — An enumeration which states if the dispatched ***Operation*** will be synchronous or asynchronous.

- synchronous — Indicates that the caller waits for the completion of the execution of the ***Operation***.
- asynchronous — Indicates that the caller does not wait for the completion of the execution of the ***Operation*** but continues immediately.

CreateAction

A *create action* is an action resulting a creation of an instance of some classifier.

In the metamodel, the ***CreateAction*** is a subtype of ***Action.*** The ***Classifier*** class is designated by the *instantiation* association of the ***CreateAction***.

Associations

classifier — The ***Classifier*** of which an ***Instance*** will be created of when the ***CreateAction*** is performed.

DestroyAction

A *destroy action* is an action results in the destruction of an object specified in the action.

In the metamodel a ***DestroyAction*** is a subclass of ***Action.*** The designated object is specified by the *target* association of the ***Action***.

DataValue

A *data value* is an instance with no identity.

In the metamodel ***DataValue*** is a subclass of ***Instance*** which cannot change its state, i.e. all ***Operations*** that are applicable to it are pure functions or queries. ***DataValues*** are typically used as attribute values.

Exception

An *exception* is a signal raised by behavioral features typically in case of execution faults. In the metamodel, ***Exception*** is derived from ***Signal***. An ***Exception*** is associated with the ***BehavioralFeature*** that raises it.

Attributes

body A description of the ***Exception*** in a format not defined in UML.

Associations

behavioralFeature The set of ***BehavioralFeatures*** that raise the exception**.**

Instance

The *instance* construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations.

In the metamodel ***Instance*** is connected to at least one ***Classifier*** which declares its structure and behavior. It has a set of attribute values and is connected to a set of ***Links***, both sets matching the definitions of its ***Classifiers***. The two sets implements the current state of the ***Instance***. ***Instance*** is an abstract metaclass.

Associations

attributeLink The set of ***AttributeLinks*** that holds the attribute values of the ***Instance***.

linkEnd The set of ***LinkEnds*** of the connected ***Links*** that are attached to the ***Instance***.

classifier The set of ***Classifiers*** that declare the structure of the ***Instance***.

Link

The *link* construct is a connection between instances.

In the metamodel ***Link*** is an instance of an ***Association.*** It has a set of ***LinkEnds*** that matches the set of ***AssociationEnds*** of the ***Association***. A ***Link*** defines a connection between ***Instances.***

Associations

association The ***Association*** the is the declaration of the ***Link***.

linkRole The sequence of ***LinkEnds*** that constitute the ***Link***.

LinkEnd

A *link end* is an end point of a link.

In the metamodel ***LinkEnd*** is the part of a ***Link*** that connects to an ***Instance***. It corresponds to an ***AssociationEnd*** of the ***Link's Association***.

Associations

instance The ***Instance*** connected to the ***LinkEnd***.

associationEnd The ***AssociationEnd*** that is the declaration of the ***LinkEnd***.

LinkObject

A *link object* is a link with its own set of attribute values and to which a set of operations may be applied.

In the metamodel ***LinkObject*** is a connection between a set of ***Instances***, where the connection itself may have a set of attribute values and to which a set of ***Operations*** may be applied. It is a subclass of both ***Object*** and ***Link***.

LocalInvocation

A *local invocation* is a special type of action that invokes a *local* operation (an operation on "self"). This type of invocation takes place without the mediation of the state machine; i.e. it does not generate a call event. The invocation of a local utility procedure of an object is an example of a ***LocalInvocation***. In contrast, a ***CallAction*** on "self" always results in an event.

In the metamodel, ***LocalInvocation*** is associated with the ***Operation*** that it invokes through the relationship to ***Request***. The *argument* association specifies the arguments of the ***Operation*** are specified by the *argument* association. (inherited from ***Action***).

MessageInstance

A *message instance* reifies a communication between two instances.

In the metamodel ***MessageInstance*** is an instance of a subclass of a ***Request***, like ***Signal*** and ***Request***. It has a sender, a receiver, and may have a set of arguments, all being ***Instances***.

Associations

specification The ***Request*** from which the ***MessageInstance*** originates.

sender The ***Instance*** which sent the ***MessageInstance***.

receiver The ***Instance*** which receives the ***MessageInstance***.

arguments The sequence of ***Instances*** being the arguments of the ***MessageInstance***.

Object

An *object* is an instance that originates from a class.

In the metamodel ***Object*** is a subclass of ***Instance*** and it originates from at least one ***Class***. The set of ***Classes*** may be modified dynamically, which means that the set of features of the ***Object*** is changed during its life-time.

Reception

A *reception* is a declaration stating that a classifier is prepared to react to the receipt of a signal. The reception designates a signal and specifies the expected behavioral response. A reception is a summary of expected behavior; the details of handling a signal are specified by a state machine.

In the metamodel ***Reception*** is a subclass of ***BehavioralFeature*** and declares that the ***Classifier*** containing the feature reacts to the signal designated by the reception feature. The *isPolymorphic* attribute specifies whether the behavior is polymorphic or not; a *true* value indicates that the behavior is not always the same and may be affected by state or subclassing. The specification indicates the expected response to the signal.

Attributes

isPolymorphic Whether the response to the *Signal* is fixed. If true, then the response may depend on state of the *Classifier* and may be overridden on subclasses. If false, then response to the signal is always the same, regardless of state of the *Classifier,* and it may not be overridden by subclasses.

specification A description of the effects of the classifier receiving a signal, stated as an ***Expression***.

Associations

signal The ***Signal*** that the ***Classifier*** is prepared to handle.

Request

A *request* is a specification of a stimulus being sent to instances. It can either be an operation or a signal.

In the metamodel a ***Request*** is an abstract subclass of ***BehavioralFeature***.

ReturnAction

A *return action* is an action that results in returning a value to a caller.

In the metamodel ***ReturnAction*** values are represented as the arguments inherited from an ***Action***.

SendAction

A *send action* is an action that results in the (asynchronous) sending of a signal. The signal can be directed to a set of receivers via objectSetExpression, or sent implicitly to an unspecified set of receivers, defined by some external mechanism. For example, if the signal is an exception, the receiver is determined by the underlying runtime system mechanisms.

In the metamodel ***SendAction*** is associated with the ***Signal*** by the *request* association inherited

from ***Action***. The actual arguments are specified by the *argument* association, inherited from ***Action***.

.

Signal

A *signal* is a specification of an asynchronous stimulus communicated between instances. The receiving instance handles the signal by a state machine. Signal is a generalizeable element and is defined independently of the classes handling the signal. A reception is a declaration that a class handles a signal, but the actual handling is specified by a state machine.

In the metamodel ***Signal*** is a subclass of ***Request*** that is dispatched by a ***SendAction***. It is a ***GeneralizableElement***, and aggregates a set of ***Parameters***. A ***Signal*** is always asynchronous.

Associations

reception	A set of ***Receptions*** that indicate *Classes* prepared to handle the signal.

TerminateAction

A *terminate action* results in self-destruction of an object.

In the metamodel ***TerminateAction*** is a subclass of ***Action***.

UninterpretedAction

An *uninterpreted action* represents all actions that are not explicitly reified in the UML

Taken to the extreme, any action is a call or raise on some instance (e.g. Smalltalk). However, in more practical terms, actions such as assignments and conditional statements can be captured as uninterpreted actions, as well as any other language specific actions that are neither call nor send actions

Attributes

body	The definition of the action.

8.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Common Behavior package.

AttributeLink

[1] The type of the ***Instance*** must match the type of the ***Attribute***.

```
self.value.classifier->includes(self.attribute.type)
```

CallAction

[1] The types and order of actual arguments for an ***Action*** must match the parameters of the ***Request.***

```
(self.actualArgument->size > 0)
```

```
      implies (Sequence{1..self.actualArguments->size})->
         forAll (x |
            self.actualArgument->at(x).type =
            self.message.parameter->at(x).type)
Note: parameter referes to Signal or Operation (downcast)
```

[2] A ***CallAction*** must have exactly one target

```
self.target->size = 1
```

[3] The type of the dispatched ***Request*** should be ***Operation***.

```
self.message->notEmpty
and
self.message.oclIsTypeOf(Operation)
```

CreateAction

[1] A ***CreateAction*** does not have a target expression.

```
self.target->isEmpty
```

DestroyAction

[1] A ***DestroyAction*** should not have arguments

```
self.actualArgument->size = 0
```

DataValue

[1] A ***DataValue*** originates from exactly one ***Classifier***, which is a ***DataType***.

```
(self.classifier->size = 1)
and
self.classifier.oclIsKindOf(DataType)
```

[2] A ***DataValue*** has no ***AttributeLinks***.

```
self.slot->isEmpty
```

Instance

[1] The ***AttributeLinks*** matches the declarations in the ***Classifiers***.

```
self.slot->forAll ( al |
   self.classifier->exists ( c |
      c.allAttributes->includes ( al.attribute ) ) )
```

[2] The ***Links*** matches the declarations in the ***Classifiers***.

```
self.allLinks->forAll ( l |
   self.classifier->exists ( c |
      c.allAssociations->includes ( l.association ) ) )
```

[3] If two ***Operations*** have the same signature they must be the same.

```
self.classifier->forAll ( c1, c2 |
   c1.allOperations->forAll ( op1 |
      c2.allOperations->forAll ( op2 |
         op1.hasSameSignature (op2)  implies op1 = op2 ) ) )
```

[4] There are no name conflicts between the ***AttributeLinks*** and opposite ***LinkEnds***.

```
self.slot->forAll( al |
   not self.allOppositeLinkEnds->exists( le | le.name = al.name ) )
and
self.allOppositeLinkEnds->forAll( le |
   not self.slot->exists( al | le.name = al.name ) )
```

[6] The number of associated ***Instances*** in one opposite ***LinkEnds*** must match the multiplicity of that ***AssociationEnd***.

Additional operations

[1] The operation *allLinks* results in a set containing all ***Links*** of the ***Instance*** itself.

```
allLinks : set(Link);
allLinks = self.linkEnd->collect ( l | l.link )
```

[2] The operation *allOppositeLinkEnds* results in a set containing all ***LinkEnds*** of ***Links*** connected to the ***Instance*** with another ***LinkEnd***.

```
allOppositeLinkEnds : set(Link);
allOppositeLinkEnds = self.allLinks->collect ( l |
   l.linkRole )->select ( le | le.instance <> self )
```

Link

[1] The set of ***LinkEnds*** must match the set of ***AssociationEnds*** of the ***Association***.

```
Sequence {1..self.linkRole->size}->forAll ( i |
  self.linkRole->at (i).associationEnd = self.association.connection->at (i) )
```

[2] There are not two ***Links*** of the same ***Association*** which connects the same set of ***Instances*** in the same way.

```
self.association.instance->forAll ( l |
  Sequence {1..self.linkRole->size}->forAll ( i |
     self.linkRole.instance = l.linkRole.instance ) implies self = l )
```

LinkEnd

[1] The type of the ***Instance*** must match the type of the ***AssociationEnd***.

```
self.instance.classifier->includes (self.associationEnd.type)
```

LinkObject

[1] One of the ***Classifiers*** must be the same as the ***Association***.

```
self.classifier->includes(self.association)
```

[2] The ***Association*** must be a kind of ***AssociationClass***.

```
self.association.oclIsKindOf(AssociationClass)
```

MessageInstance

[1] The type of the arguments must match the parameters of the ***Request***.

```
self.argument->size = self.specification.parameter->size
and
Sequence {1..self.argument->size}->forAll ( i |
   self.argument->at (i).classifier->includes (
      self.specification.parameter->at (i).type ) )
-- Note: parameter refers to the parameter of the operation or signal
```

```
-- subclasses of request.
```

Object

[1] Each of the ***Classifiers*** must be a kind of ***Class***.

```
self.classifier->forAll ( c | c.oclIsKindOf(Class))
```

Signal

[1] A ***Signal*** is always asynchronous and is always an invocation.

```
self.isAsynchronous and self.direction = activation
```

Reception

[1] A ***Reception*** can not be a query.

```
not self.isQuery
```

Request

Additional operations

[1] The *parameter* of a ***Request*** is the parameter of the ***Signal*** or Operation.

```
parameter  : set(Parameter);
parameter = if self.oclIsKindOf(Operation)
            then self.oclAsType(Operation).parameter
            else if self.oclIsKindOf(Signal)
                  then self.oclAsType(Signal).parameter
                  else Set {}
            endif endif
```

SendAction

[1] The types and order of actual arguments must match the parameters of the ***Request*** (***Signal*** or ***Operation***).

```
(self.actualArgument->size > 0)
   implies (Sequence{1..self.actualArgument->size}->
      forAll (x |
         self.actualArgument->at(x).type =
         self.message.parameters->at(x).type))
-- note: parameters apply to signal or operation (downcast)
```

[2] The type of the dispatched ***Request*** is a ***Signal***.

```
self.message->notEmpty
and
self.message.oclIsKindOf (Signal)
```

[3] The target of an ***Exception*** should be empty (implicit)

```
self.message.oclIsKindOf(Exception) implies (self.target = NULL)
```

TerminateAction

[1] A ***TerminateAction*** should not have arguments.

```
self.actualArgument->size = 0
```

8.4 SEMANTICS

This section provides a description of the semantics of the elements in the Common Behavior package.

Object and DataValue

An *object* is an instance that originates from a class—it is structured and behaves according to its class. All objects originating from the same class are structured in the same way, although each of them has its own set of attribute links. Each attribute link references an instance, usually a data value. The number of attribute links with the same name fulfils the multiplicity of the corresponding attribute in the class. The set may be modified according to the specification in the corresponding attribute, e.g. each referenced instance must originate from (a subtype of) the type of the attribute, and attribute links may be added or removed according to the changeable property of the attribute.

An object may have multiple classes, i.e. it may originate from several classes. In this case, the object will have all the features declared in all of these classes, both the structural and the behavioral ones. Moreover, the set of classes (i.e., the set of features that the object conforms to) may vary over time. New classes may be added to the object and old ones may be detached. This means that the features of the new classes are dynamically added to the object, and the features declared in a class which is removed from the object are dynamically removed from the object. No name clashes between attributes links and opposite link ends are allowed, and each operation which is applicable to the object should have a unique signature.

Another kind of instance is *data value*, which is an instance with no identity. Moreover, a data value cannot change its state—all operations that are applicable to a data value are queries and do not cause any side effects. Since it is not possible to differentiate between two data values that appear to be the same, it becomes more of a philosophical issue whether there are several data values representing the same value or just one for each value—it is not possible to tell. In addition, a data value cannot change its type.

Link

A *link* is a connection between instances. Each link is an instance of an association, i.e. a link connects instances of (subclasses of) the associated classifiers. In the context of an instance an *opposite end* defines the set of instances connected to the instance via links of the same association and each instance is attached to its link via a link-end originating from the same association end. However, to be able to use a particular opposite end, the corresponding link end attached to the instance must be navigable. An instance may use its opposite ends to access the associated instances. An instance can communicate with the instances of its opposite ends and also use references to them as arguments or reply values in communications.

A *link object* is a special kind of link, it is at the same time also an object. Since an object may change it classes this is also true for a link object. However, one of the classes must always be an association class.

Request, Signal, Exception and Message Instance

A *request* is a specification of a communication between instances as a result of an instance performing certain kinds of actions: call action, raise action, destroy action, and return action.

Two kinds of requests exist: *signal* and *operation*. The former is used to trigger a reaction in the

receiver in an asynchronous way and without a reply, and the latter is the specification of an operation, which can be either synchronous or asynchronous and may require a reply from the receiver to the sender. When an instance communicates with another instance a *message instance* is passed between the two instances. It has a sender, a receiver, and possibly a set of arguments according to the specifying request. A signal may be attached to a classifier, which means that instances of the classifier will be able to receive that signal. This is facilitated by declaring a *reception* by the classifier.

An *exception* is a special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. Unlike other signals, the receiver is determined implicitly by the interaction sequence during execution and is not explicitly specified.

The reception of a message instance originating from a call action by an instance causes the invocation of an operation on the receiver. The receiver executes the method that is found in the full descriptor of the class that corresponds to the operation. The reception of a signal by an instance may cause a transition and subsequent effects as specified by the state machine for the classifier of the recipient. This form of behavior is described in the State Machines package. Note that the invoked behavior is described by methods and state machine transitions; *Operations* and *Receptions* merely declare that a classifier accepts a given *Request* but they do not specify the implementation.

Action

An *action* is a specification of a computable statement. Each kind of action is defined as a subclass of action. The following kinds of actions are defined:

- *send action* is an action in which a message instance is created that causes a signal event for the receiver(s).
- *call action* is an action in which a message instance is created that causes an operation to be invoked on the receiver.
- *local invocation* is an action that leads to the local execution of an operation.
- *create action* is an action in which an instance is created based on the definitions of the specified set of classifiers.
- *terminate action* is an action in which an instance causes itself to cease to exist.
- *destroy action* is an action in which an instance causes another instance to cease to exist.
- *return action* is an action that returns a value to a caller.
- *uninterpreted action* is an action that has no interpretation in UML.

Each action has a specification of the target object set, which resolves into zero or more instances when the action is executed. These instances are the recipients of a signal or an operation invocation. Each action also has a list of expressions, which resolve into a list of actual argument values when the action is executed. An action is always executed within the context of an instance.

An action may dispatch a request to another instance (e.g. *call action*, *send action*). The action specifies how the receiver and the arguments are to be evaluated for each dispatched instance of the request. Moreover, the action also specifies how many message instances should be dispatched and if they should be dispatched sequentially or in parallel (*recurrence*). In a degenerated case, this could be used for specification of a condition, which must be fulfilled if the request is to be sent; otherwise, the request is neglected.

8.5 STANDARD ELEMENTS

The predefined stereotypes, constraints and tagged values for the Common Behavior package are listed in Table 4 and defined in the *Standard Elements* appendix.

Table 4: Common Behavior—Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Instance			persistent
LinkEnd		association, global, local, parameter, self	
Request		broadcast, vote	

9. BEHAVIORAL ELEMENTS PACKAGE: COLLABORATIONS

9.1 OVERVIEW

The Collaborations package is a subpackage of the Behavioral Elements package. It specifies the concepts needed to express how different elements of a model interact with each other from a structural point of view. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

A ***Collaboration*** defines a specific way to use the ***Model Elements*** in a ***Model***. It describes how different kinds of ***Classifiers*** and their ***Associations*** are to be used in accomplishing a particular task. The ***Collaboration*** defines a restriction of, or a projection of, a ***Model*** of ***Classifiers***, i.e. what properties ***Instances*** of the participating ***Classifiers*** must have in a particular ***Collaboration***. The same ***Classifier*** or ***Association*** can appear in several ***Collaborations***, and also several times in one ***Collaboration***, each time in a different role. In each appearance it is specified which of the properties of the ***Classifier*** or ***Association*** are needed in that particular usage. These properties are a subset of all the properties of that ***Classifier*** or ***Association***. A set of ***Instances*** and ***Links*** conforming to the participants specified in the ***Collaboration*** cooperate when the specified task is performed. Hence, the ***Classifier*** structure implies the possible collaboration structures of conforming ***Instances***. A ***Collaboration*** may be presented in a diagram, either showing the restricted views of the participating ***Classifiers*** and ***Associations***, or by showing prototypical ***Instances*** and ***Links*** conforming to the restricted views.

Collaborations can be used for expressing several different things, like how use cases are realized, actor structures of ROOM, OORam role models, and collaborations as defined in Catalysis. They are also used for setting up the context of ***Interactions*** and for defining the mapping between the specification part and the realization part of a ***Subsystem***.

An ***Interaction*** defined in the context of a ***Collaboration*** specifies the details of the communications that should take place in accomplishing a particular task. It describes which ***Requests*** should be sent and their internal order.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Collaborations package.

9.2 ABSTRACT SYNTAX

The abstract syntax for the Collaborations package is expressed in graphic notation in Figure 15.

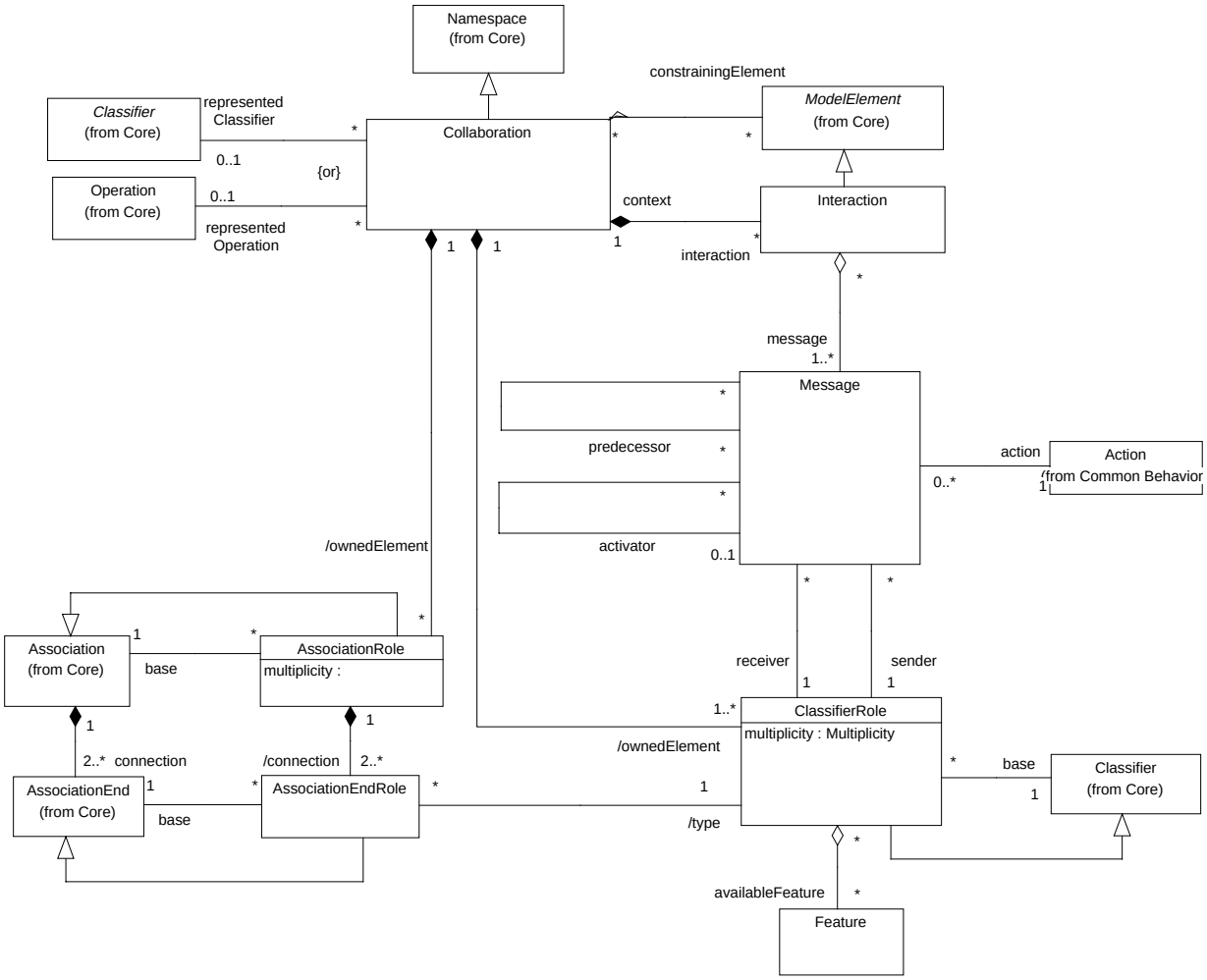


Figure 15: Collaborations

The following metaclasses are contained in the Collaborations package:

AssociationEndRole

An *association-end role* specifies an endpoint of an association as used in a collaboration.

In the metamodel an ***AssociationEndRole*** is part of an ***AssociationRole*** and specifies the connection of an ***AssociationRole*** to a ***ClassifierRole***. It is related to the ***AssociationEnd***, declaring the corresponding part in an ***Association***.

Attributes

multiplicity The number of ***LinkEnds*** playing this role in a ***Collaboration***.

Associations

base An ***AssociationEndRole*** is a projection of an ***AssociationEnd.***

AssociationRole

An *association role* is a specific usage of an association needed in a collaboration.

In the metamodel an ***AssociationRole*** specifies a restricted view of an ***Association*** used in a ***Collaboration***. An ***AssociationRole*** is a composition of a set of ***AssociationEndRoles*** corresponding to the ***AssociationEnds*** of its base ***Association***.

Attributes

multiplicity The number of ***Links*** playing this role in a ***Collaboration***.

Associations

base An ***AssociationRole*** that is a projection of an ***Association.***

ClassifierRole

A *classifier role* is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.

In the metamodel a ***ClassifierRole*** specifies one participant of a ***Collaboration***, i.e. a role ***Instances*** conform to. It declares a set of ***Features***, which is a subset of those available in the base ***Classifier***. The ***ClassifierRole*** may be connected to a set of ***AssociationRoles*** via ***AssociationEndRoles***.

Attributes

multiplicity The number of ***Instances*** playing this role in a ***Collaboration***.

Associations

availableFeature The subset of ***Features*** of the ***Classifier*** which is used in the ***Collaboration***

base A ***ClassifierRole*** that is a projection of a ***Classifier.***

Collaboration

A *collaboration* describes how an operation or a classifier, like a use case, is realized by a set of classifiers and associations used in a specific way. The collaboration defines a context for performing tasks defined by interactions.

In the metamodel a ***Collaboration*** contains a set of ***ClassifierRoles*** and ***AssociationRoles***, which represent the ***Classifiers*** and ***Associations*** that take part in the realization of the associated ***Classifier*** or ***Operation***. The ***Collaboration*** may also contain a set of ***Interactions*** that are used for describing the behavior performed by ***Instances*** conforming to the participating ***ClassifierRoles***.

A ***Collaboration*** specifies a view (restriction, slice, projection) of a model of ***Classifiers***. The projection describes the required relationships between ***Instances*** that conform to the participating ***ClassifierRoles***, as well as the required subset of the ***Features*** of these ***Classifiers***. Several

Collaborations may describe different projections of the same set of ***Classifiers***. Hence, a ***Classifier*** can be a base for several ***ClassifierRoles***.

A ***Collaboration*** may also reference a set of ***ModelElements***, usually ***Classifiers*** and ***Generalizations***, needed for expressing structural requirements, such as ***Generalizations*** required between the ***Classifiers*** themselves to fulfill the intent of the ***Collaboration***.

Associations

constrainingElement The ***ModelElements*** that add extra constraints, like ***Generalization*** and ***Constraint***, on the ***ModelElements*** participating in the ***Collaboration***.

interaction The set of ***Interactions*** that are defined within the ***Collaboration***.

ownedElement (Inherited from *Namespace*) The set of roles defined by the *Collaboration.* These are *ClassifierRoles* and *AssociationRoles.*

representedClassifier The ***Classifier*** the ***Collaboration*** is a realization of. (Used if the ***Collaboration*** represents a ***Classifier***.)

representedOperation The ***Operation*** the ***Collaboration*** is a realization of. Used if the ***Collaboration*** represents an ***Operation***.)

Interaction

An *interaction* specifies the messages sent between instances performing a specific task. Each interaction is defined in the context of a collaboration.

In the metamodel an ***Interaction*** contains a set of ***Messages*** specifying the communication between a set of ***Instances*** conforming to the ***ClassifierRoles*** of the owning ***Collaboration***.

Associations

context The ***Collaboration*** which defines the context of the ***Interaction***.

message The ***Messages*** that specify the communication in the ***Interaction***.

Message

A *message* defines how a particular request is used in an interaction.

In the metamodel a ***Message*** defines a particular usage of a ***Request*** in an ***Interaction***. It specifies the roles of the sender and receiver as well as the dispatching ***Action***. Furthermore, it defines the relative sequencing of ***Messages*** within the ***Interaction***.

Associations

activator The ***Message*** that called the operation whose method contains the current ***Message***.

base The specification of the ***Message***.

receiver The role of the ***Instance*** that receives the *Message* and reacts to it.

predecessor	The set of ***Messages*** whose completion enables the execution of the current *Message*. All of them must be completed before execution begins. Empty if this is the first message in a method.
sender	The role of the ***Instance*** that sends the *Message* and possibly receives a response.

9.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Collaborations package.

AssociationEndRole

[1] The type of the ***ClassifierRole*** must conform to the type of the base ***AssociationEnd***.

```
self.type = self.base.type
or
self.type.allSupertypes->includes (self.base.type)
```

[2] The type must be a kind of ***ClassifierRole***.

```
self.type.oclIsKindOf (ClassifierRole)
```

AssociationRole

[1] The ***AssociationEndRoles*** must conform to the ***AssociationEnds*** of the base ***Association***.

```
Sequence{ 1..(self.role->size) }->forAll (index |
   self.role->at(index).base = self.base.connection->at(index))
```

[2] The endpoints must be a kind of ***AssociationEndRoles***.

```
self.role->forAll( r | r.oclIsKindOf (AssociationEndRole) )
```

ClassifierRole

[1] The ***AssociationRoles*** connected to the ***ClassifierRole*** must match a subset of the ***Associations*** connected to the base ***Classifier***.

```
self.allAssociations->forAll( ar |
   self.base.allAssociations->exists ( a | ar.base = a ) )
```

[2] The ***Features*** of the ***ClassifierRole*** must be a subset of those of the base ***Classifier***.

```
self.base.allFeatures->includesAll (self.availableFeature)
```

[3] A ***ClassifierRole*** does not have any ***Features*** of its own.

```
self.allFeatures->isEmpty
```

Collaboration

[1] All ***Classifiers*** and ***Associations*** of the ***ClassifierRoles*** and ***AssociationRoles*** in the ***Collaboration*** should be included in the namespace owning the ***Collaboration***.

```
self.ownedElement->forAll ( e |
(e.oclIsKindOf (ClassifierRole) implies
   self.namespace.allContents->includes (e.oclAsType(ClassifierRole).base) )
and
(e.oclIsKindOf (AssociationRole) implies
   self.namespace.allContents->includes (e. oclAsType(AssociationRole).base) ))
```

[2] All the constraining ***ModelElements*** should be included in the namespace owning the ***Collaboration***.

```
self.constrainingElement->forAll ( ce |
   self.namespace.allContents->includes (ce) )
```

[3] If a ***ClassifierRole*** or an ***AssociationRole*** does not have a name then it should be the only one with a particular base.

```
self.ownedElement->forAll ( p |
   (p.oclIsKindOf (ClassifierRole) implies
      p.name = '' implies
         self.ownedElement->forAll ( q |
            q.oclIsKindOf(ClassifierRole) implies
               (p.oclAsType(ClassifierRole).base =
                  q.oclAsType(ClassifierRole).base implies p = q) ) )
and
   (p.oclIsKindOf (AssociationRole) implies
      p.name = '' implies
         self.ownedElement->forAll ( q |
            q.oclIsKindOf(AssociationRole) implies
               (p.oclAsType(AssociationRole).base =
                  q.oclAsType(AssociationRole).base implies p = q) ) )
)
```

[4] A ***Collaboration*** may only contain ***ClassifierRoles*** and ***AssociationRoles***.

```
self.ownedElement->forAll ( p |
   p.oclIsKindOf (ClassifierRole) or
   p.oclIsKindOf (AssociationRole) )
```

Interaction

[1] All ***Signals*** being bases of ***Messages*** must be included in the namespace owning the ***Interaction***.

```
self.message->forAll ( m |
   m.base.oclIsKindOf(Signal) implies
            self.collaboration.namespace.allContents->includes (m.base) )
```

Message

[1] The sender and the receiver must participate in the ***Collaboration*** which defines the context of the ***Interaction***.

```
self.interaction.context.ownedElement->includes (self.sender)
and
self.interaction.context.ownedElement->includes (self.receiver)
```

[2] The predecessors and the activator must be contained in the same ***Interaction***.

```
self.predecessor->forAll ( p | p.interaction = self.interaction )
and
self.activator->forAll ( a | a.interaction = self.interaction )
```

[3] The predecessors must have the same activator as the ***Message***.

```
self.allPredecessors->forAll ( p | p.activator = self.activator )
```

[4] A ***Message*** cannot be the predecessor of itself.

```
not self.allPredecessors->includes (self)
```

Additional operations

[1] The operation *allPredecessors* results in the set of all ***Messages*** that precede the current one.

```
allPredecessors : Set(Message);
allPredecessors = self.predecessor->union (self.predecessor.allPredecessors)
```

9.4 SEMANTICS

This section provides a description of the semantics of the elements in the Collaborations package. It is divided into two parts: Collaboration and Interaction.

Collaboration

In the following text the term *instance of a collaboration* denotes the set of instances that together participate in and perform one specific collaboration.

The purpose of a *collaboration* is to specify how an operation or a classifier, like a use case, is realized by a set of classifiers and associations. Together, the classifiers and their associations participating in the collaboration conform to the requirements of the realized operation or classifier. The collaboration defines a context in which the behavior of the realized element can be specified in terms of interactions between the participants of the collaboration. Thus, while a model describes a whole system, a collaboration is a slice, or a projection, of that model. It defines a subset of its contents, like classifiers and associations.

A collaboration may be presented at two different levels: specification level or instance level. A diagram presenting the collaboration at the specification level will show classifier roles and association roles, while a diagram at the instance level will present instances and links conforming to the roles in the collaboration.

In a collaboration it is specified what properties instances must have to be able to take part in the collaboration, i.e. each participant specifies the required set of features a conforming instance must have. Furthermore, the collaboration also states which associations must exist between the participants. Not all features of the participating classifiers and not all associations between these classifiers are always required in a particular collaboration. Because of this, a collaboration is not actually defined in terms of classifiers, but *classifier roles*. Thus, while a classifier is a complete description of instances, a classifier role is a description of the features required in a particular collaboration; i.e. a classifier role is a projection of a classifier in the sense that its features match a subset of the classifier's features. The represented classifier is referred to as the *base classifier*. Several classifier roles may have the same base classifier, even in the same collaboration, but their features may be different subsets of the features of the classifier. These classifier roles then specify different *roles* played by (usually different) instances of the same classifier.

In a collaboration the *association roles* defines what associations are needed between the classifiers in this context. Each association role represents the usage of an association in the collaboration, and it is defined between the classifier roles that represents the associated classifiers. The represented association is called the *base association* of the association role.

An instance participating in a collaboration instance plays a specific role, i.e. conforms to a classifier role, in the collaboration. The number of instances that should play one specific role in one instance of a collaboration is specified by the classifier role (*multiplicity*). Different instances may play the same role but in different instances of the collaboration. Since all these instances play the same role, they must all conform to the classifier role specifying the role. Thus, every instance must have attribute values corresponding to the attribute specified by the classifier role, and must participate in links corresponding to the association roles connected to the classifier role. The instances may, of course, have e.g. more attribute values than required by the classifier role which would be the case if they originate from a classifier being a subtype of the required one.

Furthermore, one instance may play different roles in different instances of one collaboration; the instance may, in fact, play multiple roles in the same instance of a collaboration.

If the collaboration represents an operation the context could also include things like parameters, attributes and classifiers contained in the classifier owning the operation, etc. The interactions then specify how the arguments, the attribute values, the instances etc. will cooperate to perform the behavior specified by the operation. A collaboration can be used to specify how an operation or a classifier, like a use case, is realized by a set of cooperating classifiers. In a collaboration representing an operation, the base classifiers are the operation's parameter types together with the attribute types of the classifier owning the operation. When the collaboration represents a classifier, its base classifiers can be classifiers of any kind, like classes or subsystems.

How the instances conforming to a collaboration should interact to jointly perform the behavior of the realized classifier is specified with a set of *interactions*. The collaboration thus specifies the context in which these interactions are performed.

Two or more collaborations may be *composed* in order to refine a superordinate collaboration. For example, when refining a superordinate use case into a set of subordinate use cases, the collaborations specifying each of the subordinate use cases may be composed into one collaboration, which will be a (simple) refinement of the superordinate collaboration. The composition is done by observing that at least one instance must participate in both sets of collaborating instances. This instance has to conform to one classifier role in each collaboration. In the composite collaboration these two classifier roles are merged into a new one, which will contain all features included in either of the two original classifier roles. The new classifier role will, of course, be able to fulfill the requirements of both of the previous collaborations, so the instance participating in both of the two sets of collaborating instances will conform to the new classifier role.

A collaboration may be a specification of a *template*. There will not be any instances of such a template collaboration, but it can be used for generating ordinary collaborations, which may be instantiated. Template collaborations may have parameters that act like placeholders in the template. Usually, these parameters would be classifiers and associations, but other kinds of model elements can also be defined as parameters in the collaboration, like operation or signal. In a collaboration generated from the template these parameters are refined by other model elements that make the collaboration instantiable.

Moreover, a collaboration may have a set of *constraining model elements*, like constraints and generalizations perhaps together with some extra classifiers. These constraining model elements do not participate in the collaboration, themselves. They are used for expressing extra constraints on the participating elements in the collaboration that cannot be covered by the participating roles themselves. For example, in a template it might be required that two of the classifiers must have a common ancestor or one classifier must be a subclass of another one. These kinds of requirements cannot be expressed with association roles, since they express the required links between participating instances. An extra set of model elements is therefore added to the collaboration.

Interaction

The purpose of an *interaction* is to specify the communication between a set of interacting instances performing a specific task. An interaction is defined within a *collaboration*, i.e. the collaboration defines the context in which the interaction takes place. The instances performing the communication specified by the interaction conform to the classifier roles of the collaboration.

An interaction specifies the execution of a set of *message instances*. These are partially ordered

based on which execution thread they belong to. The execution starts by executing the first message instance of each thread after it has been dispatched. Within each thread the message instances are executed in a sequential order while message instances of different threads may be executed in parallel or in an arbitrary order.

A *request* is a specification of a communication between instances, such as a call action or a send action. The request states the name of the operation to be applied to or the event to be raised in the receiver as well as the arguments. Furthermore, it specifies the direction of the stimulus, i.e. whether it is an invocation of an operation or a reply, and whether or not it is an asynchronous stimulus. If it is asynchronous the instance will continue its execution immediately after sending the message instance, while it will be blocked and waiting for a reply if it is synchronous.

A *message* is a usage of a request in an interaction. It specifies the type of the sender and the type of the receiver as well as which messages should have been received and sent before the current one. Moreover, the message also specifies the expected response of the receiver (*script*), which should be in conformance with the specification of the corresponding operation of the receiver.

The interaction specifies the *activator* and *predecessors* of each message. The *activator* is the message that invoked the procedure of which the current message is a step. Every message except the initial message of an interaction has an activator. The *predecessors* are the set of messages that must be completed before the current message may be executed. The first message in a procedure has no predecessors. If a message has more than one predecessor, then it represents the joining of two threads of control. If a message has more than one successor (the inverse of predecessor), then it indicates a fork of control into multiple threads. The *predecessors* relationship imposes a partial ordering on the messages within a procedure, whereas the *activator* relationship imposes a tree on the activation of operations. Messages may be executed concurrently subject to the sequential constraints imposed by the *predecessors* and *activator* relationship.

Each message instance is dispatched by performing an *action*. The action specifies how the receiver and the arguments are to be evaluated for each dispatched instance of the message. Moreover, the action also specifies whether iteration or conditionality should be applied and whether iteration should be applied sequentially or in parallel (*recurrence*).

9.4 STANDARD ELEMENTS

None.

9.5 NOTES

Pattern is a synonym for a template collaboration that describes the structure of a design pattern. Design patterns involve many nonstructural aspects, such as heuristics for their use and lists of advantages and disadvantages. Such aspects are not modeled by UML and may be represented as text or tables.

10.BEHAVIORAL ELEMENTS PACKAGE: USE CASES

10.1 OVERVIEW

The Use Cases package is a subpackage of the Behavioral Elements package. It specifies the concepts used for definition of the functionality of an entity like a system. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

The elements in the Use Cases package are primarily used to define the behavior of an entity, like a system or a subsystem, without specifying its internal structure**.** The key elements in this package are ***UseCase*** and ***Actor***. Instances of use cases and instances of actors interact when the services of the entity are used. How a use case is realized in terms of cooperating objects, defined by classes inside the entity, can be specified with a ***Collaboration***. A use case of an entity may be refined to a set of use cases of the elements contained in the entity. How these subordinate use cases interact can also be expressed in a ***Collaboration***. The specification of the functionality of the system itself is usually expressed in a separate use-case model, i.e. a ***Model*** stereotyped «useCaseModel». The use cases and actors in the use-case model are equivalent to those of the system package.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Use Cases package.

10.2 ABSTRACT SYNTAX

The abstract syntax for the Use Cases package is expressed in graphic notation in Figure 15.

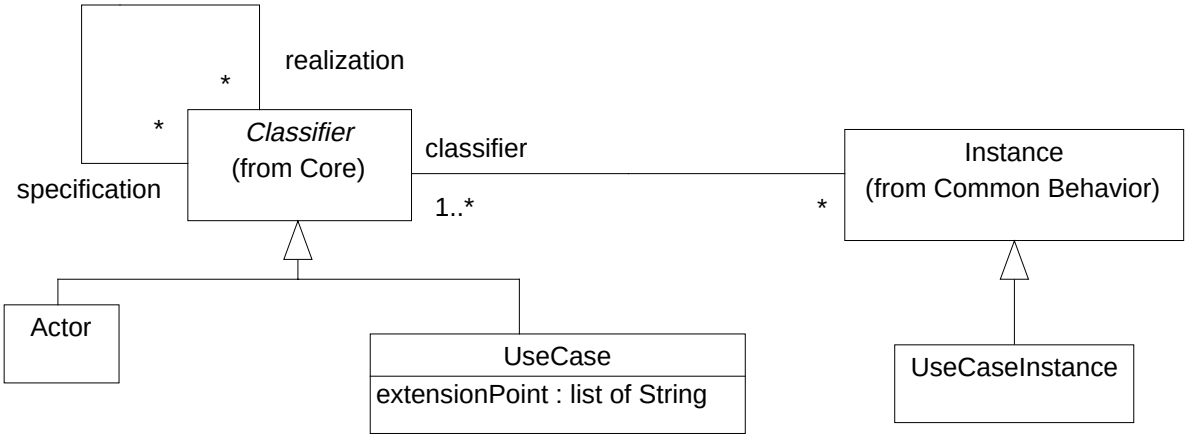


Figure 16: Use Cases

The following metaclasses are contained in the Use Cases package:

Actor

An *actor* defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor has one role for each use case with which it communicates.

In the metamodel ***Actor*** is a subclass of ***Classifier***. An ***Actor*** has a *Name* and may communicate with a set of ***UseCases***, and, at realization level, with ***Classifiers*** taking part in the realization of these ***UseCases***. An ***Actor*** may also have a set of ***Interfaces***, each describing how other elements may communicate with the ***Actor***.

An ***Actor*** may inherit other ***Actors***. This means that the inheriting ***Actor*** will be able to play the same roles as the inherited ***Actor***, i.e. communicate with the same set of ***UseCases,*** as the inherited ***Actor***.

UseCase

The *use case* construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity.

In the metamodel ***UseCase*** is a subclass of ***Classifier***, containing a set of ***Operations*** and ***Attributes*** specifying the sequences of actions performed by an instance of the ***UseCase***. The actions include changes of the state and communications with the environment of the ***UseCase***.

There may be ***Associations*** between ***UseCases*** and the ***Actors*** of the ***UseCases.*** Such an ***Association*** states that instances of the ***UseCase*** and a user playing one of the roles of the ***Actor*** communicate with each other. ***UseCases*** may be related to other ***UseCases*** only by ***Extends*** and ***Uses*** relationships, i.e. ***Generalizations*** stereotyped «**extends**» or «**uses**». An ***Extends*** relationship

denotes the extension of the sequence of one ***UseCase*** with the sequence of another one, while ***Uses*** relationships denote that ***UseCases*** share common behavior.

The realization of a ***UseCase*** may be specified by a set of ***Collaborations***, i.e. the ***Collaborations*** define how ***Instances*** in the system interact to perform the sequence of the ***UseCase***.

Attributes

extensionPoint	A list of strings representing extension points defined within the use case. An extension point is a location at which the use case can be extended with additional behavior.

UseCaseInstance

A *use case instance* is the performance of a sequence of actions being specified in a *use case*.

In the metamodel ***UseCaseInstance*** is a subclass of ***Instance***. Each method performed by a ***UseCaseInstance*** is performed as an atomic transaction, i.e. it is not interrupted by any other ***UseCaseInstance***.

An explicitly described ***UseCaseInstance*** is called a *scenario*.

10.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Use Cases package.

Actor

[1] ***Actors*** can only have ***Associations*** to ***UseCases*** and ***Classes*** and these ***Associations*** are binary.

```
self.associations->forAll(a |
   a.connection->size = 2 and
   a.allConnections->exists(r | r.type.oclIsKindOf(Actor)) and
   a.allConnections->exists(r |
      r.type.oclIsKindOf(UseCase) or
      r.type.oclIsKindOf(Class)))
```

[2] ***Actors*** cannot contain any ***Classifiers***.

```
self.contents->isEmpty
```

[3] For each ***Operation*** in an offered ***Interface*** the ***Actor*** must have a matching ***Operation***.

```
self.specification.allOperations->forAll (interOp |
self.allOperations->exists ( op | op.hasSameSignature (interOp) ) )
```

UseCase

[1] ***UseCases*** can only have binary ***Associations***.

```
self.associations->forAll(a | a.connection->size = 2)
```

[2] ***UseCases*** can not have ***Associations*** to ***UseCases*** specifying the same entity.

```
self.associations->forAll(a |
   a.allConnections->forAll(s, o|
   s.type.specificationPath->isEmpty and o.type.specificationPath->isEmpty
   or
   (not s.type.specificationPath->includesAll(o.type.specificationPath) and
   not o.type.specificationPath->includesAll(s.type.specificationPath))
```

```
))
```

[3] A ***UseCase*** can only have «uses» or «extends» ***Generalizations***.

```
self.generalization->forAll(g |
   g.stereotype.name = 'Uses' or g.stereotype.name = 'Extends')
```

[4] A ***UseCase*** cannot contain any ***Classifiers***.

```
self.contents->isEmpty
```

[5] For each ***Operation*** in an offered ***Interface*** the ***UseCase*** must have a matching ***Operation***.

```
self.specification.allOperations->forAll (interOp |
   self.allOperations->exists ( op | op.hasSameSignature (interOp) ) )
```

8.3.2.1 Additional operations

[1] The operation *specificationPath* results in a set containing all surrounding ***Namespaces*** that are not instances of ***Package***.

```
specificationPath : Set(Namespace)
specificationPath = self.allSurroundingNamespaces->select(n |
   n.oclIsKindOf(Subsystem) or n.oclIsKindOf(Class))
```

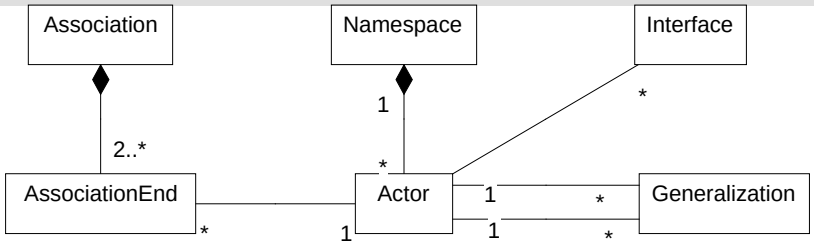
UseCaseInstance

No extra well-formedness rules.

10.4 SEMANTICS

This section provides a description of the semantics of the elements in the Use Cases package, and its relationship to other elements in the Behavioral Elements package.

Actor



Actors model parties outside an entity such as a system, a subsystem, or a class, which interact with the entity. Each actor defines a coherent set of roles users of the entity can play when interacting with the entity. Every time a specific user interacts with the entity, it is playing one such role. An instance of an actor is a specific user interacting with the entity. Any instance that conforms to an actor can act as an instance of the actor. If the entity is a system the actors represent both human users and other systems. Some of the actors of a lower level subsystem or a class may coincide with actors of the system, while others appear inside the system. The roles defined by the latter kind of actors are played by instances of classifiers in other packages or subsystems, where in the latter case the classifier may belong to either the specification part or the contents part of the subsystem.

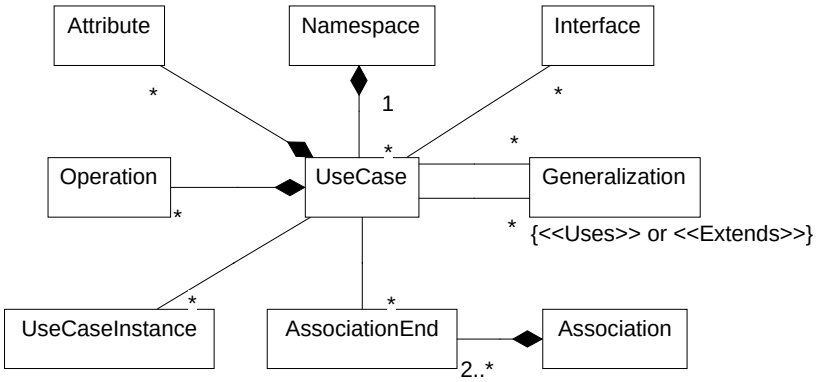
Since an actor is outside the entity, its internal structure is not defined but only its external view as

seen from the entity. Actor instances communicate with the entity by sending and receiving message instances to and from use case instances and, at realization level, to and from objects. This is expressed by associations between the actor and the use case or class.

Furthermore, *interfaces* can be connected to an actor, defining how other elements may interact with the actor.

Two or more actors may have commonalities, i.e., communicate with the same set of use cases in the same way. This commonality is expressed with *generalizations* to another (possibly abstract) actor, which models the common role(s). An instance of an heir can always be used where an instance of the ancestor is expected.

UseCase



In the following text the term *entity* is used when referring to a system, a subsystem, or a class and the term *model element* or *element* denotes a subsystem or a class.

The purpose of a *use case* is to define a piece of behavior of an entity without revealing the internal structure of the entity. The entity specified in this way may be a system or any model element that contains behavior, like a subsystem or a class, in a model of a system. Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. It specifies a complete sequence initiated by a user; i.e. the interactions between the users and the entity as well as the responses performed by the entity, as they are perceived from the outside, are specified. A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behavior, error handling etc. The complete set of use cases specifies all different ways to use the entity, i.e. all behavior of the entity is expressed by its use cases. These use cases can be grouped into *packages* for convenience.

From a pragmatic point of view, use cases can be used both for specification of the (external) requirements on an entity and for specification of the functionality offered by an (already realized) entity. Moreover, the use cases also indirectly state the requirements the specified entity poses on its users, i.e. how they should interact so the entity will be able to perform its services.

Since users of use cases always are external to the specified entity, they are represented by actors of the entity. Thus, if the specified entity is a system or a subsystem at the topmost level, i.e. a top-package, the users of its use cases are modeled by the actors of the system. Those actors of a lower level subsystem or a class that are internal to the system are often not explicitly defined. Instead, the use cases relate directly to model elements conforming to these implicit actors, i.e. whose instances play these roles in interaction with the use cases. These model elements are contained in other packages or subsystems, where in the subsystem case they may be contained in the

specification part or the contents part. The distinction between actor and conforming element like this is often neglected; thus they are both referred to by the term *actor*.

There may be associations between use cases and actors, meaning that the instances of the use case and the actor communicates with each other. One actor may communicate with several use cases of an entity, i.e. the actor may request several services of the entity, and one use case communicates with one or several actors when providing its service. Note that two use cases specifying the same entity cannot communicate with each other since each of them individually describes a complete usage of the entity. Moreover, use cases always use signals when communicating with actors outside the system, while it may use other communication semantics when communicating with elements inside the system.

The interaction between actors and use cases can be defined with *interfaces*. The interface then defines a subset of the entire interaction defined in the use case. Different interfaces offered by the same use case need not be disjoint.

A *use-case instance* is a performance of a use case, initiated by a message from an instance of an actor. As a response to the message the use-case instance performs a sequence of actions as specified by the use case, like sending messages to actor instances, not necessarily only the initiating one. The actor instances may send new messages to the use-case instance and the interaction continues until the instance has responded to all input and does not expect any more input, when it ends. Each method performed by a use-case instance is performed as an atomic transaction, i.e. it is not interrupted by any other use-case instance.

A use case can be described in plain text, using operations, in activity diagrams, by a state-machine, or by other behavior description techniques, such as pre-and post conditions. The interaction between the use case and the actors can also be presented in collaboration diagrams.

In the case where subsystems are used to model the package hierarchy, the system can be specified with use cases at all levels, since use cases can be used to specify each subsystem and each class. A use case specifying one model element is then refined into a set of smaller use cases, each specifying a service of a model element contained in the first one. The use case of the whole is said to be *superordinate* to its refining use cases, which in turn are *subordinate* to the first one. The functionality specified by each superordinate use case is completely traceable to its subordinate use cases. Note, though, that the structure of the container element is not revealed by the use cases, since they only specify the functionality offered by the element. All subordinate use cases of a specific superordinate use case cooperate to perform the superordinate one. Their cooperation is specified by *collaborations* and may be presented in collaboration diagrams. All actors of a superordinate use case appear as actors of subordinate use cases. Moreover, the cooperating subordinate use cases are actors of each other. Furthermore, the interfaces of a superordinate use case are traceable to the interfaces of those subordinate use cases that communicate with actors that are also actors of the superordinate use case.

The environment of subordinate use cases is the model element containing the model elements specified by these use cases. Thus, from a bottom-up perspective, interaction of subordinate use cases results in a superordinate use case, i.e. a use case of the container element.

Use cases of classes are specified in terms of the operations of the classes, since a service of a class in essence is the invocation of the operations of the class. Some use cases may consist of the application of only one operation, while others may involve a set of operations, possibly in a well-defined sequence. One operation may be needed in several of the services of the class, and will therefore appear in several use cases of the class.

The realization of a use case depends on the kind of model element it specifies. For example, since the use cases of a class are specified by means of operations, they are realized by the corresponding methods, while the use cases of a subsystem are realized by the elements contained in the subsystem. Since a subsystem does not have any behavior of its own, all services offered by a subsystem must be a composition of services offered by elements contained in the subsystem, i.e. eventually by classes. These elements will collaborate and jointly perform the behavior of the specified use case. One or a set of collaborations describes how the realization of a use case is made. Hence, collaborations are used for specification of both the refinement and the realization of a use case.

The usage of use cases at all levels imply not only a uniform way of specification of functionality at all levels, but also a powerful technique for tracing requirements at the system package level down to operations of the classes. The propagation of the effect of modifying a single operation at the class level all the way up to the behavior of the system package is managed in the same way.

Commonalities between use cases are expressed with *uses* relationships (i.e., generalizations with the stereotype «uses»). The relationship means that the sequence of behavior described in a used use case is included in the sequence of another use case. The latter use case may introduce new pieces of behavior anywhere in the sequence as long as it does not change the ordering of the original sequence. Moreover, if a use case has several uses relationships, its sequence will be the result of interleaving the used sequences together with new pieces of behavior. How these parts are combined to form the new sequence is defined in the using use case.

An *extends relationship*, i.e. a generalization with the stereotype «extends», defines that a use case may be extended with some additional behavior defined in another use case. The extends relationship includes both a *condition* for the extension and a reference to an *extension point* in the related use case, i.e. a position in the use case where additions may be made. Once an instance of a use case reaches an extension point to which an *extends* relationship is referring, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case. Different parts of the extending use case sequence may be inserted at different extension points in the original sequence. If there is still only one condition (i.e., if the condition of the extends relationship is fulfilled at the first extension point), then the entire extending behavior is inserted in the original sequence.

Note that the two kinds of relationships described above can only exist between use cases specifying the same entity. The reason for this is that the use cases of one entity specify the behavior of that entity alone, i.e., all use-case instances are performed entirely within that entity. If a use case would have a uses or extends relationship to a use case of another entity, the resulting use-case instances would involve both entities, resulting in a contradiction. However, uses and extends relationships can be defined from use cases specifying one entity to use cases of another one if the first entity has a generalization to the second one, since the contents of both entities are available in the first entity.

As a first step when developing a system, the dynamic requirements of the system as a whole can be expressed with use cases. The entity being specified is then the whole system, and the result is a separate model called a *use-case model* (i.e., a model with the stereotype «useCaseModel»). Next, the realization of the requirements is expressed with a model containing a system package, probably a package hierarchy, and at the bottom a set of classes. If the system package, i.e. the representation of the system as a whole in the model, is modeled by applying the «topLevelPackage» stereotype to the subsystem construct, its abstract behavior is naturally the same as that of the system. Thus, if use cases are used for the specification part of the system package, these use cases are equivalent to those in the use-case model of the system, i.e. they

express the same behavior but possibly slightly differently structured. In other words, all services specified by the use cases of a system package, and only those, define the services offered by the system. Furthermore, if several models are used for modeling the realization of a system, e.g. an analysis model and a design model, the set of use cases of all system packages and the use cases of the use-case model must be equivalent.

10.5 STANDARD ELEMENTS

See the *Standard Elements* appendix for definitions of the «extends», «extends» and «useCaseModel» stereotypes.

10.6 NOTES

A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors. This ensures that the use cases are complete specifications and not just fragments.

11. BEHAVIORAL ELEMENTS PACKAGE: STATE MACHINES

11.1 OVERVIEW

The State Machine package is a subpackage of the Behavioral Elements package. It specifies a set of concepts that can be used for modeling behavior through finite state-transition systems. It is defined as an elaboration of the Foundation package. The State Machine package also depends on concepts that are defined in the Common Behavior package, enabling integration with the other subpackages in Behavioral Elements.

The metamodel described supports an object variant of statecharts. Statecharts are characterized by a number of conceptual shortcuts, such as hierarchical states, concurrent states, history, and branch nodes, which, in combination, achieve a significant compaction of specifications over most other state-based formalisms. In a sense, all other finite-state machine models can be considered as constrained versions of statecharts (e.g., Mealy machines or state-event matrices).

State machines can be used in two different ways. In one case, the state machine may specify complete behavior of its context, typically a class. In that case requestors send requests to the owner of a state machine. and the state machine receiving an event determines what the effect will be by attaching actions to transitions, from which complete specifications of operations can be derived.

In the second case, the state machine may be used as a protocol specification, showing the order in which operations may be invoked on a type. Transitions are triggered by call events and their actions invoke the desired operation. This means that a caller is allowed to invoke the operation at that point. The protocol state machine does not specify actions that specify the behavior of the operation itself, but shows a change of state determining which operations can be invoked next.

In addition to defining state machines, the metamodel also defines the core semantics of activity models. Statecharts and activity models share many elements, and hence are based on the same metamodel. Activity models are a subtype of state models that use most of the concepts that apply to state machines.

The following sections describe the abstract syntax, well-formedness rules and semantics of the State Machines package.

11.2 ABSTRACT SYNTAX

The abstract syntax for the State Machines package is expressed in graphic notation in Figure 15.

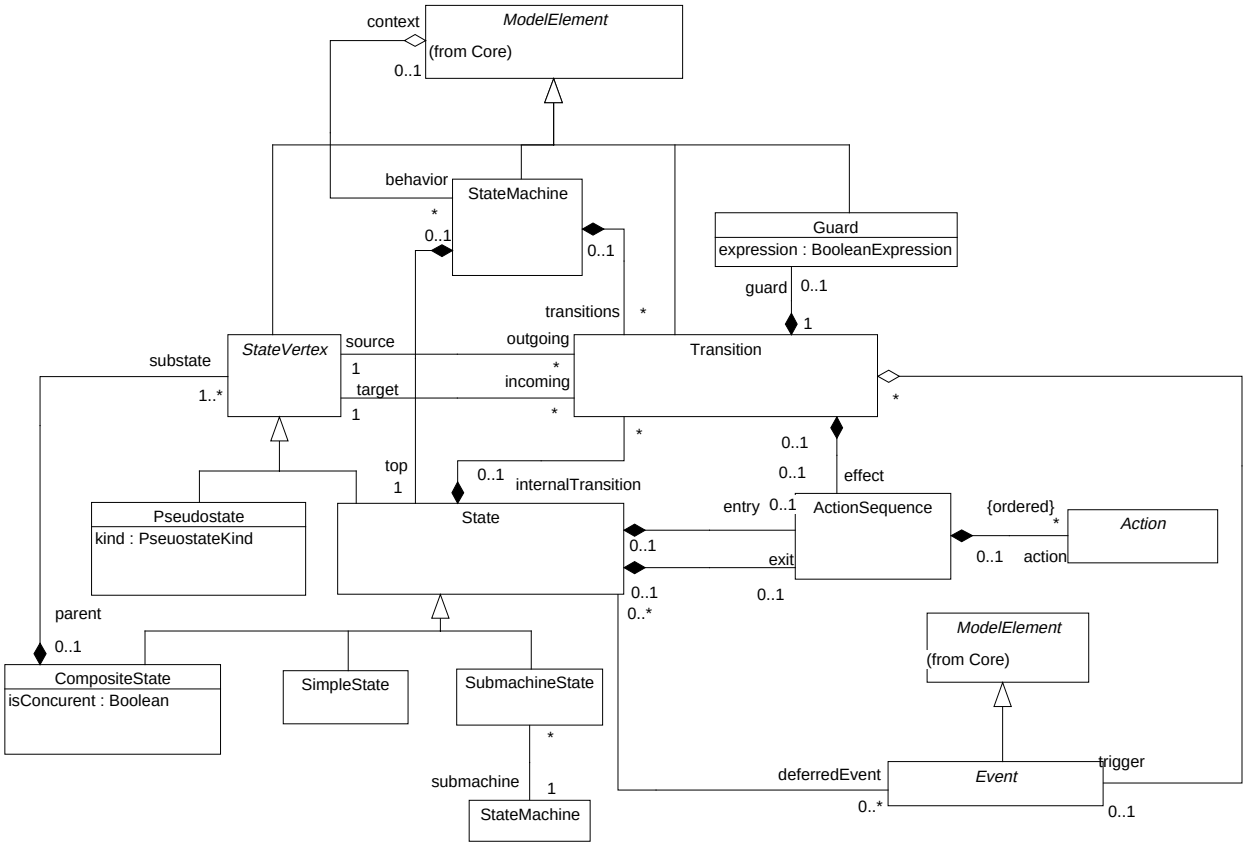


Figure 17: State Machines—Main

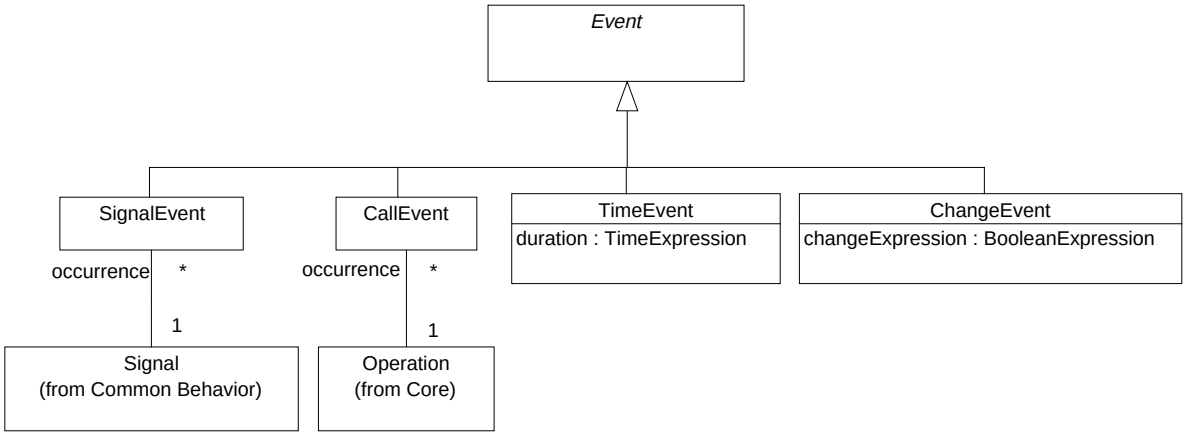


Figure 18: State Machines—Events

The following metaclasses are contained in the State Machines package:

CallEvent

A *call event* is the reception of a request to invoke an operation. The expected result is the execution of the operation.

In the metamodel ***CallEvent*** is a subclass of ***Event***, which is the abstract meta-class representing all event types that trigger a transition in the state machine.

Two special cases of ***CallEvent*** are the object creation event and the object destruction event.

Associations

operation	Designates the operation whose invocation is requested.

ChangeEvent

A *change event* is an event that is generated when one or more attributes or relationships change value according to an explicit expression.

A change event is never raised by an explicit change event action. Instead, it is a consequence of the execution of one or more actions that modify the values of elements that are referenced in the boolean expression. The corresponding change event is actually raised by the underlying run-time system that detects that the condition has changed to true

A change event functions as a trigger for transitions, and must not be confused with a guard. When a change event occurs, a guard can still block any transition that would otherwise be triggered by that change.

In the metamodel ***ChangeEvent*** is a subclass of ***Event,*** which is the abstract class that represents all events that trigger a ***StateMachine***.

Attributes

changeExpression	A boolean expression that indicates when a ***ChangeEvent*** occurs.

CompositeState

A *composite state* is a state that consists of substates.

In the metamodel a ***CompositeState*** is a subclass of ***State*** that contains one or more substates that are subtypes of ***StateVertex.***

Associations

substates	Designates a set of ***States*** that constitute the substates of a ***CompositeState***. Each substate is uniquely owned by its parent ***CompositeState***.

Attributes

isConcurrent	A boolean value that specifies the decomposition semantics. If this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive components (usually associated with concurrent execution). If this attribute is false, then

there are no direct orthogonal components in the composite.[4] This means that exactly one of the substates can be active at a given instant (i.e., sequential execution).

isRegion A derived boolean value that indicates whether a ***CompositeState*** is a substate of a concurrent state. If it evaluates to true, then the ***CompositeState*** is a substate of a concurrent state.

Event

An *event* is the specification of a significant occurrence that has a location in time and space. An instance of an event can lead to the activation of a behavioral feature in an object.

It is important to distinguish between an event, which is a static specification for a dynamically occurring concept, from an actual instance of an event as a result of program execution. The class ***Event*** represents the type of an event. An instance of an event is not modeled explicitly in the metamodel.

In the metamodel an ***Event*** is a subclass of ***ModelElement*** and is the part of a ***Transition*** that represents its *trigger*.

Guard

A *guard* condition is a boolean expression that may be attached to a transition in order to determine whether that transition is enabled or not.

The guard is evaluated when an event occurrence triggers the transition. Only if the guard is true at the time the event is presented to the state machine will the transition actually take place. Guards should be pure expressions without side effects. Guard expressions with side effects may lead to unpredictable results.

In the metamodel ***Guard*** is a ***ModelElement*** so it can be substituted in refined state machines.

Attributes

expression A boolean expression that specifies the guard condition.

PseudoState

A *pseudo state* is an abstraction of different types of nodes in the state machine graph which represent transient points in transition paths from one state to another (e.g., branch and fork points). Pseudo states are used to construct complex transitions from simple transitions. For example, by combining a transition entering a fork pseudo state with a set of transitions exiting the fork pseudo state, we get a complex transition that leads to a set of target states.

In the metamodel ***PseudoState*** is a subclass of ***StateVertex***, which generalizes all statechart nodes.

Attributes

[4] However, there may be orthogonal components in some of the contained states.

kind	Determines the type of the ***PseudoState*** and can be one of: initial, deepHistory, shallowHistory, join, fork, branch, final.

SignalEvent

A *SignalEvent* represents events that result from the reception of a signal by an object.

In the metamodel ***SignalEvent*** is a subclass of ***Event***.

Associations

signal	Designates the ***Signal*** whose reception by the state owner may trigger a ***Transition***..

SimpleState

A *SimpleState* is a state that does not have substates.

In the metamodel a ***SimpleState*** is a subclass of ***State*** that does not have any additional features. It is included solely for symmetry with ***CompositeState***.

State

A *State* is a condition or situation during the life of an object during which is satisfies some condition, performs some activity, or waits for some event. A state models a dynamic situation in which, typically, one or more (implicit or explicit) conditions hold.

In the metamodel, a ***State*** is a subclass of ***StateVertex***, thereby inheriting the fundamental features of incoming and outgoing ***transitions*** associated with state vertices.

Associations

deferredEvent	A list of ***Events*** the effect of whose occurrence during the ***State*** is postponed until the owner enters a ***State*** in which they are not deferred, at which time they may trigger ***Transitions*** as if they had just occurred.
entry	An optional ***ActionSequence*** that is executed when the ***State*** is entered. These ***Actions*** are atomic, may not be avoided, and precede any internal activity or ***Transitions***.
exit	An optional ***ActionSequence*** that is executed when the ***State*** is exited. These ***Actions*** are atomic, may not be avoided, and follow any internal activity or ***Transitions***.
internalTransition	A set of ***Transitions*** that occur entirely within the ***State***. If one of their triggers is satisfied then the action is performed without changing ***State***. This means that the entry or exit condition of the ***State*** will not be invoked. These ***Transitions*** apply even if the ***StateMachine*** is in a nested region and they leave it in the same ***State***.
deferredEvent	An association that specifies the ***Events*** to be deferred if received within the ***State***. Multiplicity '*..*' indicates that a ***State*** can defer multiple ***Events***, and an ***Event*** can be deferred by multiple ***States***.

StateMachine

A *state machine* is a behavior that specifies the sequences of states that an object or an interaction

goes through during its life in response to events, together with its responses and actions. The behavior is specified as a traversal of a graph of state nodes interconnected by one or more joined transition arcs. The transitions are triggered by series of event instances.

In the metamodel a ***StateMachine*** is composed of ***States*** and ***Transitions***. The ***ModelElement*** role provides the *context* for the ***StateMachine***. A common case of the context relation is where a ***StateMachine*** is designated to specify the *lifecycle* of the ***Classifier***. The ***StateMachine*** has a composition aggregation to a ***State*** that represents the *top* state and a set of ***Transitions***. As a consequence the ***StateMachine*** owns its ***Transitions*** and its top ***State***, but nested states are transitively owned through their parent ***States***.

Associations

context	An association to a ***ModelElement*** constrained to be a ***Classifier*** or a ***BehavioralFeature***. The owning ***ModelElement*** is the element whose behavior is specified by the ***StateMachine***. The ***ModelElement*** may contain multiple ***StateMachines*** (although for many purposes one suffices). Each ***StateMachine*** is owned by one ***ModelElement***.
top	Designates the top level ***State*** directly owned by the ***StateMachine***. Other ***States*** are owned by the parent composite states. The multiplicity is 1, there must be one ***State*** designated as the top ***State***. The rest of the ***StateMachine*** is an expansion of this ***CompositeState***.
transitions	Associates the ***StateMachine*** with its ***Transitions***. Note that internal ***Transitions*** are owned by the ***State*** and not by the ***StateMachine***. All other ***Transitions*** which are essentially relationships between ***States*** are owned by the ***StateMachine***. Multiplicity is ‘0..*’.

StateVertex

A *StateVertex* is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

In the metamodel a ***StateVertex*** is a subclass of ***ModelElement***.

Associations

outgoing	Specifies the transitions departing from the vertex.
incoming	Specifies the transitions entering the vertex.

SubmachineState

A *SubmachineState* represents a nested state machine. A nested state machine is semantically equivalent to a composite state, but facilitates reuse and modularity in the form of an independent nested state machine.

In the metamodel a ***SubmachineState*** is a subclass of ***State***.

Associations

submachine	Represents the substate machine

TimeEvent

A *TimeEvent* is a subtype of ***Event*** for modeling event instances resulting from the expiration of a deadline.

In the metamodel a time event can specify a trigger of a transition, which by default denotes the time elapsed since the current state was entered.

Attributes

duration	Specifies the corresponding time deadline.

Transition

A *Transition* is a relationship between a *source* state vertex and a *target* state vertex. It may be part of a *compound transition*, which takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event instance for a given source state configuration.

In the metamodel ***Transition*** is a subclass of ModelElement that participates in various relationships with other state machine metaclasses:

Associations

trigger	Specifies the single ***Event*** which activates it
guard	Predicate that must evaluate to true at the instant the ***Transition*** is triggered.
effect	Specifies an ***ActionSequence*** to be performed when the ***Transition*** fires.
source	Designates the ***StateVertex*** affected by firing the ***Transition***. If the ***StateVertex*** is in the source ***state*** and the trigger of the ***Transition*** is satisfied, then it fires, performs its ***Actions***, and the ***StateMachine*** enters the target ***State***.
target	Designates the ***StateVertex*** that results from a firing of the ***Transition*** when the ***StateMachine*** was originally in the source ***State***. After the firing the ***StateMachine*** is in the target ***State***.

11.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the State Machines package.

CompositeState

[1] A composite state can have at most one initial vertex

```
self.subState->select (v | v.oclType = Pseudostate)->
   select(p : Pseudostate | p.kind = #initial)->size <= 1
```

[2] A composite state can have at most one deep history vertex

```
self.subState->select (v | v.oclType = Pseudostate)->
   select(p : Pseudostate | p.kind = #deepHistory)->size <= 1
```

[3] A composite state can have at most one shallow history vertex

```
self.subState->select(v | v.oclType = Pseudostate)->
   select(p : Pseudostate | p.kind = #shallowHistory)->size <= 1
```

[4] There have to be at least two composite substates in a *concurrent* composite state

```
(self.isConcurrent) implies
   (self.subState->select (v | v.oclIsKindOf(CompositeState))->size <= 2)
```

Guard

[1] A guard should not have side effects

LocalInvocation

[1] A local invocation has no target

```
self.target->size = 0
```

PseudoState

[1] An initial vertex can have at most one outgoing transition and no incoming transitions

```
(self.kind = #initial) implies
   ((self.outgoing->size <= 1) and (self.incoming->isEmpty))
```

[2] A final pseudo state cannot have outgoing transitions

```
(self.kind = #final) implies (self.outgoing->isEmpty)
```

[3] History vertices can have at most one outgoing transition

```
((self.kind = #deepHistory) or (self.kind = #shallowHistory)) implies
   (self.outgoing->size <= 1)
```

[4] A join vertex must have at least two incoming transitions and exactly one outgoing transition

```
(self.kind = #join) implies
   ((self.outgoing->size = 1) and (self.incoming->size >= 2))
```

[5] A fork vertex must have at least two outgoing transitions and exactly one incoming transition

```
(self.kind = #fork) implies
   ((self.incoming->size = 1) and (self.outgoing->size >= 2))
```

[6] A branch vertex must have one incoming transition segment and at least two outgoing transition segments with guards.

```
(self.kind = #branch) implies
   ((self.incoming->size = 1) and
      ((self.outgoing->size >= 2) and self.outgoing->forAll(t |
            t.guard->size = 1)))
```

StateMachine

[1] A StateMachine is aggregated within either a classifier or a behavioral feature.

```
self.context.oclIsKindOf(BehavioralFeature) or self.context.oclIsKindOf(Classifier)
```

[2] A top state is always a composite

```
self.top.oclIsTypeOf(CompositeState)
```

[3] A top state cannot have parents

```
self.top.parent->isEmpty
```

[4] The top state cannot be the source or target of a transition

```
(self.top.outgoing->isEmpty) and (self.top.incoming->isEmpty)
```

[5] There can be no history vertices in the top state

```
self.top.substate->select( oclIsTypeOf(Pseudostate) )->
   forAll (p : Pseudostate |
      not (p.kind = #shallowHistory) and not (p.kind = #deepHistory))
```

[6] If a StateMachine describes a behavioral feature, it contains no triggers of type CallEvent, apart from the trigger on the initial transition (see OCL for Transition [8]).

```
self.context.oclIsKindOf(BehavioralFeature) implies
self.transitions->reject(
   source.oclIsKindOf(Pseudostate) and
         source.oclAsType(Pseudostate).kind= #initial).trigger->isEmpty
```

Transition

[1] A fork segment should not have guards or triggers

```
self.source.oclIsKindOf(Pseudostate) implies
   ((self.source.oclAsType(Pseudostate).kind = #fork) implies
      ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[2] A join segment should not have guards or triggers

```
self.target.oclIsKindOf(Pseudostate) implies
   ((self.target.oclAsType(Pseudostate).kind = #join) implies
      ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[3] A fork segment should always target a state

```
self.source.oclIsKindOf(Pseudostate) implies
   ((self.source.oclAsType(Pseudostate).kind = #fork) implies
                        (self.target.oclIsKindOf(State)))
```

[4] A join segment should always originate from a state

```
self.target.oclIsKindOf(Pseudostate) implies
   ((self.target.oclAsType(Pseudostate).kind = #join) implies
         (self.source.oclIsKindOf(State)))
```

[5] A branch segment must not have a trigger

```
self.source.oclIsKindOf(Pseudostate) implies
   (((self.source.oclAsType(Pseudostate).kind = #branch) or
   (self.source.oclAsType(Pseudostate).kind = #deepHistory) or
   (self.source.oclAsType(Pseudostate).kind = #shallowHistory) or
   (self.source.oclAsType(Pseudostate).kind = #initial)) implies
                                          (self.trigger->isEmpty))
```

[6] Join segments should originate from orthogonal states

```
self.target.oclIsKindOf(Pseudostate) implies
   ((self.target.oclAsType(Pseudostate).kind = #join) implies
            (self.source.parent.isConcurrent))
```

[7] Fork segments should target orthogonal states

```
self.source.oclIsKindOf(Pseudostate) implies
   ((self.source.oclAsType(Pseudostate).kind = #fork) implies
         (self.target.parent.isComposite))
```

[8] An initial transition at the topmost level may have a trigger with the stereotype “create”. An initial transition of a StateMachine modeling a behavioral feature has a CallEvent trigger associated with that BehavioralFeature. Apart from these cases, an initial transition never has a trigger.

```
self.source.oclIsKindOf(Pseudostate) implies
   ((self.source.oclAsType(Pseudostate).kind = #initial) implies
      (self.trigger->isEmpty or
      ((self.source.parent = self.stateMachine.top) and
               (self.trigger.stereotype.name = 'create')) or
      (self.stateMachine.context.oclIsKindOf(BehavioralFeature) and
               self.trigger.oclIsKindOf(CallEvent) and
               (self.trigger.oclAsType(CallEvent).operation =
                                          self.stateMachine.context))
      ))
self.source.oclIsKindOf(Pseudostate) implies
   ((self.source.kind = #initial) implies
      (self.trigger.isEmpty or
      ((self.source.parent = self.StateMachine.top) and
               (self.trigger.stereotype.name = 'create')) or
      (self.StateMachine.context.oclIsKindOf(BehaviouralFeature) and
               self.trigger.oclIsKindOf(CallEvent) and
               (self.trigger.operation = self.StateMachine.context))
      ))
```

11.4 SEMANTICS

This section describes the execution semantics of state machines. For convenience, the semantics are described using an operational style; that is, they are expressed in terms of the operations of a hypothetical machine that implements a state machine specification. In the general case, the key components of this abstract machine are:

- an events queue which accepts incoming event instances,
- a dispatcher which selects and de-queues event instances for processing, and
- an event processor which processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question. Because of that, this component is simply referred to as “the state machine” in the following text.

This is for reference purposes only and is not meant to imply that individual realizations must conform to this structure. For example, the role of the event dispatcher might be played by some other object that simply invokes an operation on the object.

Understanding the dynamic semantics of state machines requires an understanding of the complex relationships between individual metaclasses. Therefore, the bulk of the description of the dynamic semantics of state machine is included in the context of the state machine metaclass.

StateMachine

The software context that we assumes that a state machine reacts to an event applied to it by some external object.

Event processing by a state machine is partitioned into *steps*, each of which is caused by an event instance directed to the state machine.

The fundamental semantics assumes that events are processed in sequence, where each event stimulates a *run-to-completion* (RTC) step. The next external event is dispatched to the state machine after the previous RTC step has completed. This assumption simplifies the transition function of the state machine since the incoming event is processed only after the state machine has reached a well-defined (stable) *state configuration*.

The practical meaning of these semantics is *thread protection*, allowing the state machine to safely complete its RTC step without concern about being interrupted in mid-transition by a subsequent event. This may be implemented by a thread event-loop reading events from a queue (in case of active classes) or as a monitor (in case of a passive class).

It is possible to define state machine semantics by allowing the RTC steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics as defined in this document are based on the precept that an RTC step applies to the entire state machine. This satisfies most practical purposes.

Run-to-completion processing

Once an event instance is dispatched, it may result in one or multiple transitions being *enabled* for firing. (Only transitions that triggered by the corresponding event type can be enabled.) . By default, if no transition is enabled, the event is discarded without any effect. An event can be *deferred* to be processed later if specified as a deferred event in one of the active states. Deferred events semantics are described in a following section.

In case where one or more transitions are enabled, the state machine selects a subset and *fires* them, moving the state machine from one active state configuration to a new active state configuration. This basic transformation is called a *step*. The transitions that fire are determined by the *transition selection function* described below. Actions that result from taking the transition may cause event instances to be generated for this and other objects.

If these actions are synchronous then the transition *freezes* until the invoked objects complete their own run. Each orthogonal bottom-level component can fire at most one transition as a result of the event instance dispatch. Conflicting transitions (described below) will not fire in the same step. When all orthogonal regions have finished executing the transition, the event instance is consumed, and the step terminates.

The order in which selected transitions fire is not defined. It is based on an arbitrary traversal that is not explicitly defined by the state machine formalism.

Completion transitions and completion events

A completion transition is a transition without a trigger (a guard is possible). The completion transition is typically taken upon the completion of actions of its source state.

After reacting to an event occurrence, the state machine may reach a state configuration where some of the states have outgoing completion transitions (transient configurations). Such a configuration is considered non-stable.

In this case further steps are taken until the state machine reaches a stable state configuration (i.e., no more transitions are enabled). Completion transitions are triggered by completion events, which are dispatched to the state machine whenever a transient configuration is encountered. Completion events are dispatched in a series of steps until a stable configuration is reached completing the RTC step initiated by the event instance. At this point, control returns to the dispatcher and a new event instance can be dispatched.

It is possible for a state machine to never reach a stable configuration. (A practical solution to overcome such cases in an implementation of this semantics, is to set a limit on the maximal number of steps allowed before the state machine is to reach a stable configuration.)

An event instance can arrive at a state machine that is *frozen* in the middle of an RTC step from some other object within the same thread, in a circular fashion. This event instance can be treated by orthogonal components of the state machine that are not frozen along transitions at that time.

Step semantics

Informally, the semantics of a step involve the execution of a maximal set of non-conflicting transitions from an active, current state configuration. (Note that this section is based on the dynamic semantics sections of ***State***, ***CompositeState***, and ***Transition***.)

Transition selection

Transition selection specifies which subset of the enabled transitions will fire. The following sections discuss the two major considerations that affect transition selection: conflicts and priorities.

Conflicts

In a given state, it is possible for several transitions to be enabled within a state machine. The issue then is which ones can be fired simultaneously without contradicting (conflicting with) each other. For example, if there are two transitions originating from a state s, one labeled e[c1] and the other e[c2], and if both [c1] and [c2] are true, then only one transition can fire.

Two transitions are said to *conflict* if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. The intuition is that only ‘concurrent’ transitions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

An internal transition in a state conflicts only with transitions that cause an exit from that state.

Priorities

Priorities resolve transition conflicts, but not all of them. We use the state hierarchy to define priorities among conflicting transitions. By definition, a transition emanating from a substate has higher priority than a conflicting transition emanating from any of the containing states.

The priority of a transition is defined based on its source state. Join transitions get the priority

according to their lowest source state.

If t1 is a transition whose source state is s1, and t2 has source s2, then:

- If s1 is a substate of s2, then t1 has higher priority than t2.
- If s1 and s2 are not hierarchically related, then there is no priority defined between t1 and t2.

(Note that other policies are also possible. In classical statecharts, the priority is reversed: parent states imply higher priorities than nested states. However, in the object context inner states are more specialized than their ancestors, and therefore override them.)

Selecting transitions

The set of transitions that will fire is the maximal set that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicts within the set.
- There is no transition outside the set that has higher priority than a transition in the set. Intuitively, the ones with higher priorities are in the set and the ones with lower priorities are left out.

This definition is not written algorithmically, but can be easily implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. Active states are traversed bottom up, where transitions originating from each state are evaluated. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels. This is resolved by "locking" each orthogonal state once a transition inside any one of its components is fired. The bottom-up traversal and the orthogonal state locking together guarantee a proper selection set.

Deferred events

Each of the states in the active states configuration may specify a set of deferred events. In case where no transition is enabled following an event dispatch, if the event is specified to be deferred by any of the active configuration states, it is considered pending.

An event instance is pending as long as its event is deferred by the active configuration. Following an RTC step where the state machine reaches a configuration in which the event is not deferred, the event instance is *ready* to be dispatched again.

Note the it is the responsibility of the dispatching mechanism to serialize the events to be dispatched in a sequence, since the step semantics is assumes a single event dispatch. Therefore if following an RTC-step more than a single pending event becomes ready (or an external event has occurred) it is guaranteed that there is no conflict.

State

During execution a state can be *active* or *inactive*. A state becomes active when it is entered as a result of some transition, and becomes inactive if it is exited as a result of a transition.

A state can be exited and entered as a result of the same transition (e.g., self transition).

Whenever a state is entered, it executes its entry action sequence. Whenever a state is exited, it executes its exit action sequence.

CompositeState

Legal state configuration

Every *active* composite state during execution must follow the legal active state configuration with respect to its substates. This means that the following constraints are always met during execution (except for transition execution period which is transient):

- If the composite state is not a concurrent state, exactly one of its substates is active.
- If the composite state is concurrent, all of its substates (regions) are active.

To avoid violation of the legal configuration constraints during execution, the dynamic semantics upon entering and exiting composite states is defined such that a well-formed state machine always satisfies them.

Entering a composite state

Entering a non-concurrent composite state

Upon entering a composite state the entry action sequence executes similar to simple state.

- *default entry*: If the transition hits the edge of the composite state, then the default (initial[5]) transition executes to enter one of the substates of the composite state. Note that initial transitions must always be enabled (in case of branches). A disabled initial transition is an ill-defined execution state and its handling is an implementation issue.
- *explicit entry*: If the transition "passes through" the state towards one of its substates, then the explicit substate becomes active, and recursively follows the entering procedure.
- *history entry*: if the transition is entering a history pseudo state of a composite state, the active substate is determined as the most recent active substate prior to the entry. If it is the first time the state is entered, then the active substate is determined by the transition outgoing from the history pseudo state. If no such transition is specified, the situation is illegal and its resolution is implementation dependent. The active substate determined by history proceeds with its default entry.

[5] Initial transition is a transition originating from an initial pseudo state.

- *deep history entry*: similar to history, but the active substate also executes deep history entry (recursively)

Entering a concurrent composite state

Whenever a concurrent composite state is entered, each one of its substates (the “regions”) are also entered, either by default or explicitly. If the transition hits the edge of the composite state, then all the regions are default entered. If the transition explicitly enters one or more regions (fork), these regions are entered explicitly and the others by default.

Exiting a composite state

Exiting non-concurrent state

The active substate(s) is exited (recursively). After exiting the active substate, the exit action is executed.

Exiting a concurrent state

Each one of the regions is exited. Following that, the exit actions are executed.

Pseudostate

A ***Pseudostate*** represents family of nodes in the state machine that are attached to states and transitions as compositional elements that carry additional semantics.

A Pseudostate can be one of the following:

- *initial* represents a default vertex that is the source for a single transition to the “default” state. There can be at most one initial vertex in a composite state or state machine.

- *deepHistory* is a vertex that is used to represent, in shorthand form, the most recent active configuration of a state and its substates. A composite state can have at most one history vertex. A transition coming into the history vertex is equivalent to a transition coming into the most recent active configuration of a state and the transitive closure of all its substates. A transition originating from the history connector leads to the default history state. This transition is taken in case no history exists and a transition to history is taken.

- *shallowHistory* is a vertex that is used to represent, in shorthand form, the most recent active configuration of a state but not its substates. A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. (Note that a state can have both deepHistory and shallowHistory transitions.)

- *join* vertices combine several transition segments coming from source vertices in different orthogonal components. The segments entering a join vertex cannot have guards.

- *fork* vertices connect an incoming transition to two or more orthogonal target vertices. The segments outgoing from a fork vertex must not have guards.

- *branch* vertices split a single segment into two or more transition branches labeled by guards. The guards determine which of the branches are enabled. A predefined guard denoted “else” may be defined for at most one branch. This branch is enabled if all the guards labeling the other branches are false.

- *final* represents a simple state with some additional semantics. Unlike all other pseudo states, this is not a transient state. When the final state is entered, its parent composite state is terminated, or that it satisfies the termination condition. In case that the parent of the final state is the top state, the entire statechart terminates, and this implies the termination of “life” of the entity that the statechart specifies. If the statechart specifies the behavior of a classifier, it implies the “termination” of that instance. In case that the parent state of the final state is not the top state, it simply means that the terminate transitions are enabled.

  A terminate transition is a transition is a transition outgoing a non-pseudo state which does not have a label (event or guard). It is enabled if its source state has reached a final state.

SubmachineState

A *submachine state* is an organizational concept and does not introduce additional behavioral semantics. The submachine state facilitates reuse of state machine segments similar to the way procedures and templates are used in conventional programming language. A submachine state also facilitates decomposition of complex state machines into a set of simpler machine.

The semantics of a submachine state is equivalent to the semantics of replacing the submachine state with the state machine related by the submachine association, where the top state of the submachine merges with the submachine state, resulting in a composite state. Therefore, it is possible that the submachine state has entry or exit actions and/or internal transitions, they are attached to the resulting CompositeState.

A submachine state may also be thought of as a state machine “subroutine”, in which one machine “calls” another machine and then “returns” to the original machine.

Transitions

Transitions vs. compound transitions

In the general case a *transition* represents a fragment of a *compound transition*. A compound transition is a cluster of simple transitions connected by *join*, *fork*, and *branch* transitions. In case of branch nodes, only one segment is selected for each branch, based on the guard. The dynamic semantics specify the execution of a compound transition, which is atomic in terms of execution (join, fork, and branch are pseudostates, not states).

Note that a compound transition can have at most one trigger, since join, fork and branch segments cannot have triggers.

A transition that fires always leads from one legal state configuration to another legal state configuration. Transitions originating from a composite state, once fired, always cause exiting the composite state and its constituents.

High-level (“interrupt”) transitions

Transitions originating from composite states are sometimes referred to as “high-level” transitions or “interrupts”. Once selected to fire (as explained below), they result in exiting of all the internal substates and executing their exit actions. Note however, that since the state machine semantics are run-to-completion, strictly speaking they are not really interrupts, but rather generalized or “group” transitions. (The term “interrupt” stems from classical statecharts where so-called “do activities” of states would be aborted as a result of high-level transitions.)

Enabled (compound) transitions

A transition is *enabled* if both of the following hold:

- All source states of the transition are in the current active state configuration. A completion transition (without a trigger) requires its source state to be in the termination state, in case it is a composite state.

- The trigger matches the event instance posted to the state machine. Null triggers match any event, in particular completion event. A specialized event matches a trigger based on a generalized event .

- There is a path of transition segments from the source to the target states, along which all the guards are satisfied (transition without guards are always satisfied). If more than one path is possible, only one is selected (non-deterministically).

Note that guards are evaluated prior to the invocation of any action related to the transition.

Since guards are not interpreted, their evaluation may include expressions causing side effects. Guards causing side effects are considered bad practice, since their evaluation strategy, in terms of when guards are evaluated and in which order, is not defined and is a function of the implementation.

(Compound) Transition execution

Transition execution semantics are defined such that the resulting state configuration is always a legal one. This principle is especially important once we deal with transitions entering/exiting boundaries of concurrent states.

LCA, main source, and main target

Every compound transition causes the exit of one (composite) state, and proper entering of another composite state. These two states are designated as the ***main source*** and the ***main target*** of the transition.

The Least Common Ancestor (LCA) state of a transition is the lowest state that contains all the explicit source states and explicit target states of the compound transition. In case of branch segments, only the states related to the selected path are considered explicit targets (“dead” branches are not considered).

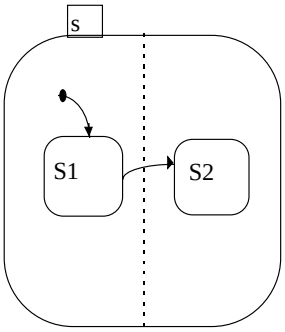
The *main source* is a direct substate of the *LCA* that contains the explicit sources. The *main target* is a substate of the *LCA* that contains the explicit targets.

Examples:

1. The common simple case: A transition *t* between two simple states *s1* and *s2*, in a composite state *s*.

   Here *LCA*(t) is *s*, the main source is *s1* and the main target is *s2*.

2. A more esoteric case: An unstructured transition from one region to another.



Here *LCA*(t) is the parent of *s*, the main source is *s* and the main target is *s*.

Transition execution sequence

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly exited (as defined in the composite states exiting semantics above).
- Actions are executed in sequence following their linear order along the segments of the transition: The "closer" the action to the source state, the earlier it is executed.
- The main target state is properly entered (as defined in the composite state entry semantics above).

11.5 STANDARD ELEMENTS

The predefined stereotypes, constraints and tagged values for the State Machines package are listed in Table 5 and defined in the *Standard Elements* appendix.

Table 5: State Machines—Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Event	«create» «destroy»		

11.6 NOTES

Example: Modeling class behavior

In the software that is implemented as a result of a state modeling design, the state machine may or may not be actually visible in the (generated or hand-crafted) code. The state machine will not be visible if there is some kind of run-time system that supports state machine behavior. In the more general case, however, the software code will contain specific statements that implement the state machine behavior.

A C++ example is shown in Figure 19:

```
class bankAccount {

private:

int balance;

public:

void deposit(amount)
{
  if (balance > 0)  balance = balance + amount; // no charge

  else

  balance = balance + amount - 1 ; // $1 charge for the transaction
}

void withdrawal(amount) {

if (balance>0) balance = balance - amount ;

}

}
```

Figure 19: Modeling Class Behavior Example

In the above example, the class has an abstract state manifested by the *balance* attribute, controlling the behavior of the class. This is modeled by the state machine in Figure 20.

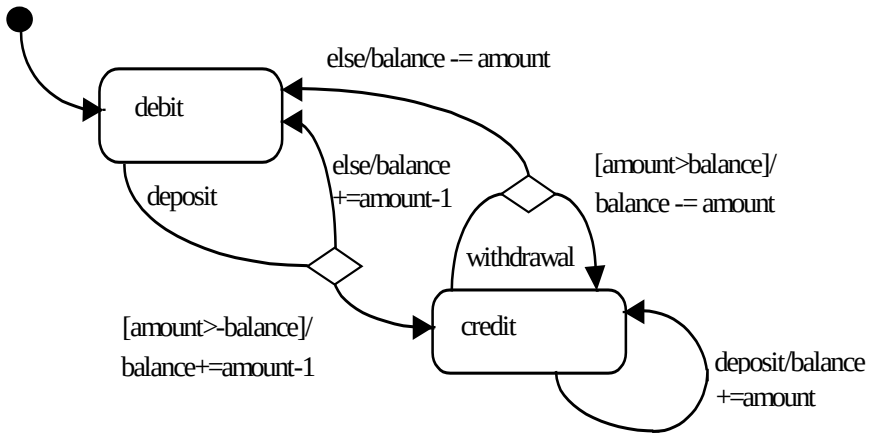


Figure 20: State Machine for Modeling Class Behavior

Since state machines describe behaviors of generalizable elements, primarily classes, state machine refinement is used capture the relationships between the corresponding state machines. The refinement mechanism itself is part of the Auxiliary Elements package, and define general refinement relationships between arbitrary model composites.

Example: State machine refinement

Since state machines describe behaviors of generalizable elements, primarily classes, state machine refinement is used capture the relationships between the corresponding state machines. The refinement relationships are facilitated by the refinement metaclass defined in the auxiliary elements package. State machines use refinement in three different mappings, specified by the mapping attribute of the refinement meta-class. The mappings are refinement, substitution, and deletion.

To illustrate state machine refinement, consider the following example where one state machine attached to a class denoted ‘Supplier’, is refined by another state machine attached to a class denoted as ‘Client’.

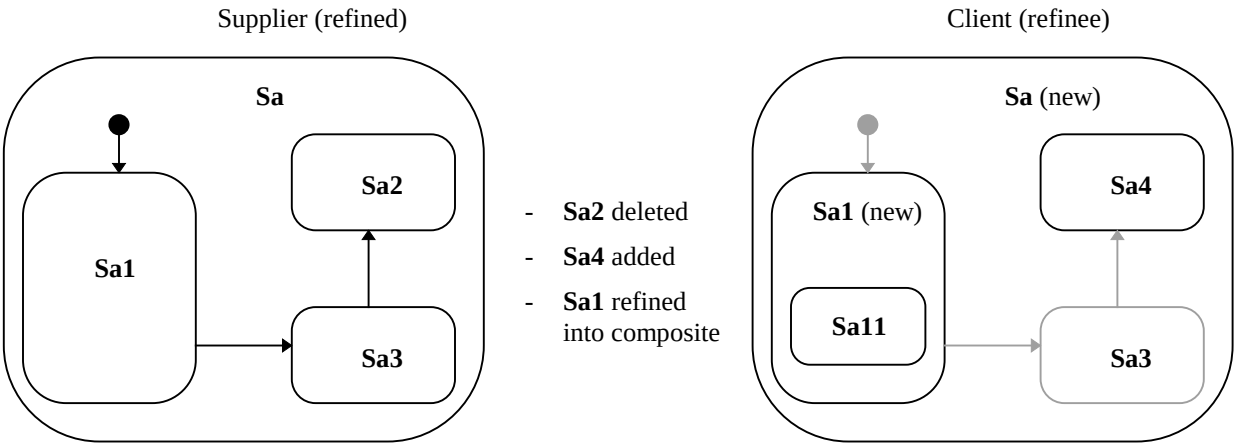


Figure 21: State Machine Refinement Example

In the example above, the client state (**Sa**(new)) in the subclass substitutes the simple substate (**Sa1**) by a composite substate (**Sa1**(new)). This new composite substate has a component substate (**Sa11**). Furthermore, the new version of **Sa1** deletes the substate **Sa2** and also adds a new substate **Sa4**. Substate **Sa3** is inherited and is therefore common to both versions of **Sa**. For clarity, we have used a gray shading to identify components that have been inherited from the original. (This is for illustration purposes and is not intended as a notational recommendation.)

It is important to note that state machine refinement as defined here does not specify or favor any specific policy of state machine refinement. Instead, it simply provides a flexible mechanism that allows subtyping, (behavioral compatibility), inheritance (implementation reuse), or general refinement policies.

We provide a brief discussion of potentially useful policies that can be implemented with the state machine refinement mechanism. These policies could be indicated by attaching standard stereotypes (i.e. «subtype» and «inherits») to the refinement relationship between state machines.

Subtyping

The refinement policy for subtyping is based on the rationale that the subtype preserves the pre/post condition relationships of applying events/operations on the type, as specified by the state machine. The pre/post conditions are realized by the states, and the relationships are realized by the transitions. Preserving pre/post conditions guarantee the substitutability principle.

States and transitions are only added, not deleted. Refinement is interpreted as follows:

- A refined State has the same outgoing transitions, but may add others, but a different set of incoming transitions. It may have a bigger set of substates, and it may change its concurrency property from false to true.

- A refined Transition may go to a new target state which is a substate of the state specified in the base class. This comes to guarantee the post condition specified by the base class.

- A refined Guard has the same guard condition, but may add disjunctions. This guarantees that pre-conditions are weakened rather than strengthened.

- A refined ActionSequence contains the same actions (in the same sequence), but may have additional actions. The added actions should not hinder the invariant represented by the target state of the transition.

(Strict) Inheritance

The rationale behind this policy is to encourage reuse of implementation rather than preserving behavior. Since most implementation environment utilize strict inheritance (i.e. features can be replaced or added, but not deleted), the inheritance policy follows this line by disabling refinements which may lead to non-strict inheritance once the state machine is implemented.

States and transitions can be added. Refinement is interpreted as follows:

- A refined State has some of the same incoming transitions (i.e. drop some, add some) but a greater or bigger set of outgoing transitions. It may have more substates, and may change its concurrency attribute.

- A refined Transition may go to a new target state but should have the same source.
- A refined Guard has may have a different guard condition
- A refined ActionSequence contains some of the same actions (in the same sequence), and may have additional actions

General Refinement

In this most general case, states and transitions can be added and deleted (i.e. 'null' refinements). Refinement is interpreted without constraints, i.e. there are no formal requirements on the properties and relationships of the refined state machine element, and the refining element:

- A refined State may have different outgoing and incoming transitions (i.e. drop all, add some)
- A refined Transition may leave from a different source and go to a new target state
- A refined Guard has may have a different guard condition
- A refined ActionSequence need not contain the same actions (or it may change their sequence), and may have additional actions

The refinement of the composite state in the example above is an illustration of general refinement.

It should be noted that if a type has multiple supertype relationships in the structural model, then the default state machine for the type consists of all the state machines of its supertypes as orthogonal state machine regions. This may be explicitly overridden through refinement if required.

Classical statecharts

The major difference between classical (Harel) statecharts and object state machines result from the external context of the state machine. Object state machines primarily come to represent behavior of a type. Classical statechart specify behaviors of processes. The following list of differences result from the above rationale:

- Events carry parameters, rather than being primitive signals
- Call events (operation triggers) are supported to model behaviors of types
- Event conjunction is not supported, and the semantics is given in respect to a single event dispatch, to better match the type context as opposed to a general system context.
- Classical statecharts have an elaborated set of predefined actions, conditions and events which are not mandated by object state machines, such as entered(s), exited(s), true(condition), tr!(c) (make true), fs!(c).
- Operations are not broadcast but can be directed to an object-set.

- The notion of activities (processes) does not exist in object state machines. Therefore all predefined actions and events that deal with activities are not supported, as well as the relationships between states and activities.

- Transition compositions are constrained for practical reasons. In classical statecharts any composition of pseudo states, simple transitions, guards and labels is allowed.

- Object state machine support the notion of synchronous communication between state machines.

- Actions on transitions are executed in their given order.

- Classical statecharts are based on the zero-time assumption, meaning transitions take zero time to execute. The whole system execution is based on synchronous steps where each step produces new events that will be processed at the next step. In OO state machines, this assumptions are relaxed and replaced with these of software execution model, based on threads of execution and that execution of actions do take time.

11.7 ACTIVITY MODELS

11.7.1 Overview

Activity models define an extended view of the State Machine package. State machines and activity models are both essentially state transition systems, and share many metamodel elements. This section describes the concepts in the State Machine package that are specific to activity models. It should be noted that the activity models extension has few semantics of its own—it should be understood in the context of the State Machine package, including its dependencies on the Foundation package and the Common Behavior package.

An activity model is a special case of a state machine model that is used to model processes involving one or more classifiers. Most of the states in such a model are action states that represent atomic actions, i.e., states that invoke actions and then wait for their. Transitions into action states are triggered by events, which can be 1) the completion of a previous action state; 2) the availability of an object in a certain state; 3) the occurrence of a signal; or 4) the satisfaction of some condition. By defining a small set of additional subtypes to the basic state machine concepts, the well-formedness of activity models can be defined formally, and subsequently mapped to the dynamic semantics of state machines. In addition, the activity specific subtypes eliminate ambiguities that might otherwise arise in the interchange of activity models between tools.

11.7.2 Abstract Syntax

The abstract syntax for activity models is expressed in graphic notation in Figure 22.

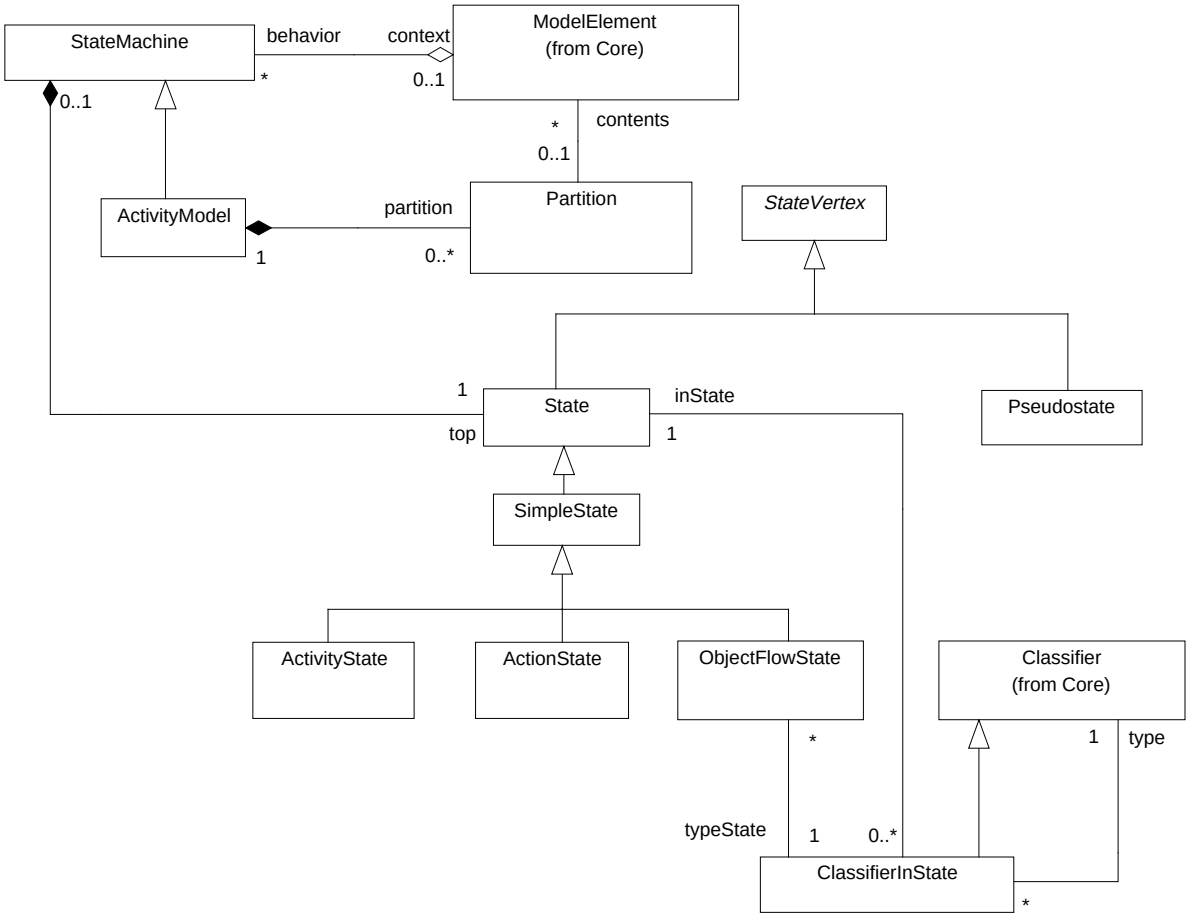


Figure 22: Activity Models

The following metaclasses are used to define activity models:

ActivityModel

An *activity model* is a special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent actions. It does not extend the semantics of state machines but it does define shorthand forms that are convenient for modeling computational processes.

The primary basis for ***ActivityModels*** is to describe a state model of an activity or process involving one or more ***Classifiers***. ActivityModels can be attached to ***Packages***, ***Classifiers*** (including ***UseCases***) and ***BehavioralFeatures***. Most of the ***States*** in an activity model are ***ActionStates***, i.e., states in which an action is being performed, typically the execution operations. As in any state machine, if an outgoing transition is not explicitly triggered by an event then it is implicitly triggered by the completion of the contained actions. An ***ActivityState*** represents structured subactivity that has some duration and internally consists of a set of actions. That is, an ActivityState is a “hierarchical action” with an embedded activity submodel that ultimately resolves to individual actions.

Ordinary “wait states” can be included to model situations in which the computation waits for an external event. Branches, forks, and joins may also be included to model decisions and concurrent activity.

ActivityModels include the concept of ***Partitions*** to organize states according to various criteria, such as the real-world organization responsible for their performance.

Activity modeling can be applied in the context of organizational modeling for business process engineering and workflow modeling. In this context, events often originate from ‘outside’ the system (e.g. ‘customer call’). Activity models can also be applied to system modeling to specify the dynamics of operations and system level processes when a full interaction model is not needed.

Associations

partition	A set of ***Partitions*** each of which contains some of the model elements of the model.

ActionState

An *action state* represents the execution of an atomic action, typically the invocation of an operation.

An ***ActionState*** is a ***SimpleState*** with an entry action whose only exit ***Transition*** is triggered by the implicit event of completing the execution of the entry action. The state therefore corresponds to the execution of the entry action itself and the outgoing ***Transition*** is activated as soon as the action has completed its execution.

An ***ActionState*** may perform more than one ***Action*** as part of its entry ***ActionSequence.*** An ***ActionState*** may not have an exit transition, internal transitions, or external transitions triggered by anything other than the implicit action completion event.

Associations

entry	(Inherited from ***State***.) Specifies the invoked actions.

ActivityState

An *activity state* represents the execution of a non-atomic sequence of steps that has some duration (i.e., internally it consists of a set of actions and possibly waiting for events). That is, an activity state is a “hierarchical action”, where an associated sub-activity model is executed.

An ***ActivityState*** is a ***SubmachineState*** that executes a nested activity model. When an input transition to the ***ActivityState*** is triggered, execution begins with the initial state of the nested ***ActivityModel.*** The outgoing ***Transition*** of an ***ActivityState*** is enabled when the final state of the nested ***ActivityModel*** is reached (i.e., when it completes its execution).

The semantics of an ***ActivityState*** are equivalent to the model obtained by statically substituting the contents of the nested model as a composite state replacing the activity state.

Associations

submachine	(Inherited from ***SubmachineState***.) Designates an activity model that is conceptually nested within the activity state. The activity state is conceptually equivalent to a ***CompositeState*** whose contents are the states of the nested ***ActivityModel***. The nested activity model must have an initial state and a final state.

ClassifierInState

A *classifier in state* characterizes instances of a given classifier for a particular state. In an activity model, it may be input and/or output to an action through an object flow state.

ClassifierInState is a subtype of ***Classifier*** and may be used in static structural models and collaborations. e.g., it can be used to show associations that are only relevant when objects of a class are in a given state.

Associations

type	Designates a ***Classifier*** that characterizes instances.
inState	Designates a ***State*** that characterizes instances. The state must be a valid state of the corresponding ***Classifier***.

ObjectFlowState

An *object flow state* defines an object flow between actions in an activity model. It signifies the availability of an instance of a classifier in a given state, usually as the result of an operation. This state indicates that an instance of the given class having the given state is available when the state is occupied.

The generation of an object by an action in an ***ActionState*** may be modeled by an ***ObjectFlowState*** that is triggered by the completion of the ***ActionState***. The use of the object in a subsequent ***ActionState*** may be modeled by connecting the output transition of the ***ObjectFlowState*** as an input transition to the ***ActionState.*** Generally each action places the object in a different state that is modeled as a distinct ***ObjectFlowState***.

Associations

typeState	Designates the class (or other classifier) and state of the object.

Partition

A *partition* is a mechanism for dividing the states of an activity model into groups. Partitions often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the states of an activity model.

Associations

contents	Specifies the states that belong to the partition. They need not constitute a nested region.

It should be noted that ***Partitions*** do not impact the dynamic semantics of the model but they help to allocate properties and actions for various purposes.

PseudoState

A *pseudo state* is an abstraction of different types of nodes in a state machine graph which function as transient points in transitions from one state to another, such as branching and forking.

Final ***PseudoStates*** are used for modeling hierarchical activities. A transition to a *final* ***PseudoState*** within an ***ActivityModel*** can be used to indicate completion of a sub-***ActivityModel*** such that execution is resumed at the superstate level (i.e. outgoing superstate transitions will be

activated). A nested activity model must have both an initial state and a final state or states.

11.7.3 Well-Formedness Rules

ActivityModel

[1] An ActivityModel specifies the dynamics of
(i) a *Package*, or
(ii) a *Classifier* (including *UseCase*), or
(iii) a *BehavioralFeature*.

```
(self.context.oclIsTypeOf(Package)    xor
 self.context.oclIsKindOf(Classifier) xor
 self.context.oclIsKindOf(BehavioralFeature))
```

[2] An ActivityModel that specifies the dynamics of a BehavioralFeature or that is nested has exactly one initial State, representing the invocation of the BehavioralFeature or subactivity.

ActionState

[1] An ActionState has exactly one outgoing Transition.

```
self.outgoing->size = 1
```

[2] An ActionState has a non-empty Entry ActionSequence.

```
self.entry.action->size > 0
```

[3] An ActionState does not have an internal Transition or an Exit ActionSequence.

```
self.internalTransition->size = 0 and self.exit->size = 0
```

ObjectFlowState

[1] The ClassifierInState of the ObjectFlowState is the type of an input Parameter to an Operation invoked in the ActionStates which have the ObjectFlowState on an incoming Transition.

```
self.outgoing.target->select(oclIsTypeOf(ActionState)).
      invoked.parameter->select(
            kind = #in or kind = #inout).type->includes(self.typeState.type)
```

[2] The ClassifierInState of the ObjectFlowState is the type of an output Parameter of an Operation invoked in the ActionStates which have the ObjectFlowState on an outgoing Transition.

```
self.incoming.source->select(oclIsTypeOf(ActionState)).
      invoked.parameter->select(
            kind = #out or kind = #inout or kind = #return).
                        type->includes(self.typeState.type)
```

PseudoState

[1] In ActivityModels, Transitions incoming to (and outgoing from) *join* and *fork* PseudoStates have as sources (targets) any StateVertex. That is, *joins* and *forks* are syntactically not restricted to be used in combination with CompositeStates, as is the case in StateMachines.

```
self.stateMachine.oclIsTypeOf(ActivityModel) implies
   ((self.kind = #join or self.kind = #fork) implies
         (self.incoming->forAll(source.oclIsKindOf(SimpleState) or
                                          source.oclIsTypeOf(PseudoState)) and
```

```
(self.outgoing->forAll(source.oclIsKindOf(SimpleState) or
                       source.oclIsTypeOf(PseudoState)))))
```

[2] All of the paths leaving a fork must eventually rejoin in a subsequent join or joins. Furthermore, if there are multiple layers of joins they must be well nested. Therefore the concurrency structure of an activity model is in fact equally restrictive as that of an ordinary state machine, even though the composite states need not be explicit.

11.7.4 Semantics

ActivityModel

The dynamic semantics of activity models can be expressed in terms of state machines. This means that the process structure of activities formally must be equivalent to orthogonal regions (in composite states). That is, transitions crossing between parallel paths (or threads) are not allowed. As such, an activity specification that contains 'unconstrained parallelism' as is used in general activity models is considered 'incomplete' in terms of UML.

All events that are not relevant in a state must be deferred so they are consumed when become relevant. This is facilitated by the general deferral mechanism of state machines.

ActionState

As soon as the incoming transition of an ActionState is triggered (either through a single transition or through an conjunction of transitions connected to a 'join'), its entry action starts executing. Once the entry action has finished executing, the action is considered completed. Hence, formally, an activated action state signifies that the execution of an action is ongoing. When the action is complete then the outgoing transition (either a simple transition or a 'fork') is enabled.

ObjectFlowState

The activation of an ObjectFlowState signifies that an instance of the associated Classifier is available in a specified State (i.e., a state change has occurred as a result of a previous operation). This may enable a subsequent action state that requires the instance as input. The execution of the action consumes the value. If the ***ObjectFlowState*** leads into a join pseudostate, then the ObjectFlowState remains activated until the other predecessors of the join have completed.

Unless there is an explicit 'fork' that creates orthogonal object states, only one of an ObjectFlowState's outgoing transitions will fire, based on the activation of the first ActionState that requires it as input. The invocation of the ActionState will generally result in a state change of the object, resulting in a new ObjectFlowState.

11.7.6 Notes

Object-flow states in activity models are a specialization of the general *dataflow* aspect of process models. Object-flow activity models extend the semantics of standard dataflow relationships in three areas:

- The operations in action states in activity models are operations of classes or types (e.g., 'Trade' or 'OrderEntryClerk'). They are not hierarchical 'functions' operating on a dataflow.

- The ‘contents’ of object flow states are typed. They are not unstructured data definitions as in data stores.

- The state of the object flowing as input and output between operations is defined explicitly. It is the event of the availability of an object in a specific state that forms a trigger for the operation that requires the object as input. Object flow states are not stateless, passive data definitions as are data stores.

PART 4. GENERAL MECHANISMS

Part 4 defines mechanisms of general applicability to models. This version of UML contains one general mechanisms package, Model Management. The Model Management package specifies how model elements are organized into models, packages and systems.

Contents

12. Model Management

12.MODEL MANAGEMENT PACKAGE

12.1 OVERVIEW

The Model Management package is a subpackage of the Behavioral Elements package. It defines ***Model***, ***Package*** and ***Subsystem***, elements that serve mainly as grouping units for other ***ModelElements***. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

Packages are used within a ***Model*** to group ***ModelElements***. A ***Subsystem*** is a special kind of ***Package*** with an additional specification of the behavior offered by ***ModelElements*** in the ***Subsystem***.

In this section the term *modeled system* denotes the physical entity being modeled with UML, i.e. the term is not one of the constructs in the modeling language. It can denote a computer system, like a *seat assignment system*, a *banking system*, or a *telephone exchange system*. It can also describe business processes, like a *sales process*, or a *development process*. An analogy with the construction of houses would be that *house* would correspond to *modeled system*, while *blue print* would correspond to *model* and *element used in a blue print* would correspond to *model element* in UML.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Model Management package.

12.2 ABSTRACT SYNTAX

The abstract syntax for the Model Management package is expressed in graphic notation in Figure 23.

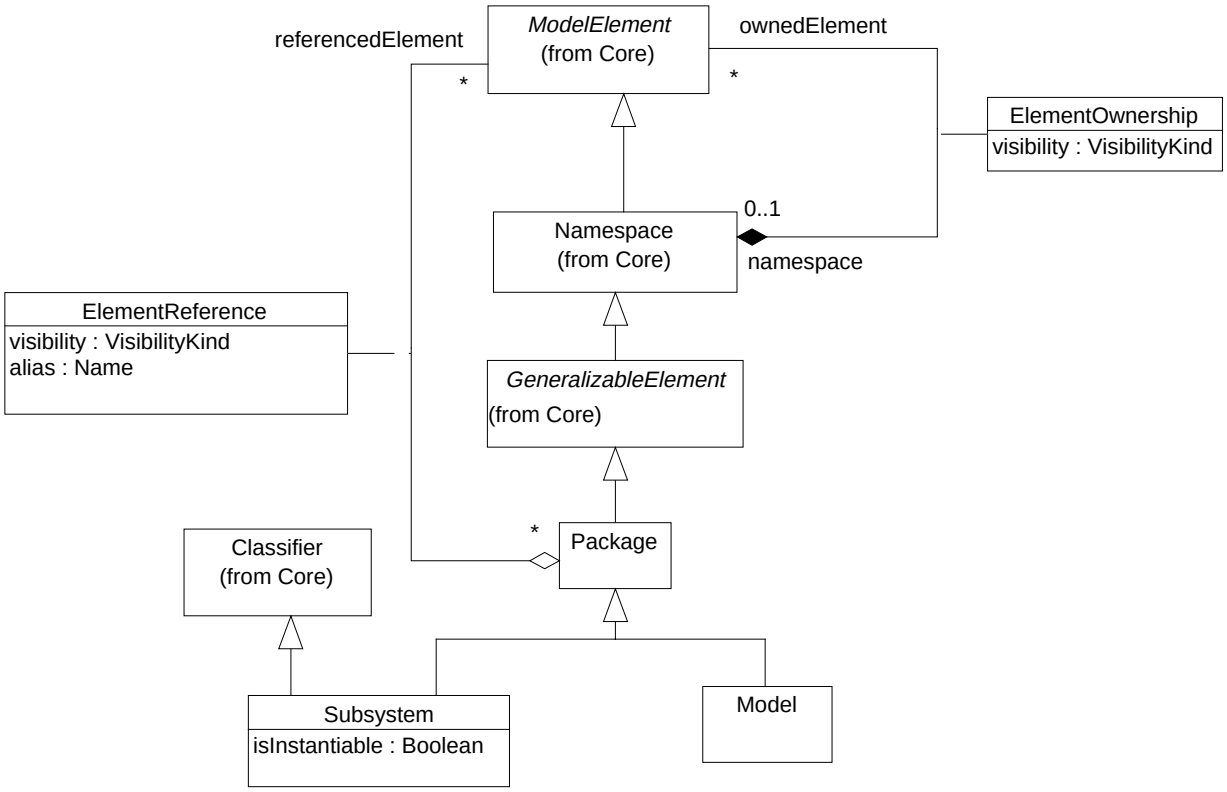


Figure 23: Model Management

The following metaclasses are contained in the Model Management package:

ElementReference

An *element reference* defines the visibility and alias of a model element referenced by a package.

In the metamodel an ***ElementReference*** reifies the relationship between a ***Package*** and a ***ModelElement***. It defines the alias for the ***ModelElement*** inside the ***Package*** and the visibility of the ***ModelElement*** relative to the ***Package***.

Attributes

alias	The alias defines a local name of the referenced ***ModelElement***, to be used within the ***Package***.
visibility	Each referenced ***ModelElement*** is either public, protected, or private relative to the referencing ***Package***.

Associations

- No extra associations.

Model

A *model* is an abstraction of a modeled system, specifying the modeled system from a certain viewpoint and at a certain level of abstraction. A model is complete in the sense that it fully describes the whole modeled system at the chosen level of abstraction and viewpoint.

In the metamodel ***Model*** is a subclass of ***Package***. It contains a containment hierarchy of ***ModelElements*** that together describe the modeled system. A ***Model*** also contains a set of ***ModelElements***, like ***Actors***, which represents the environment of the system, together with their interrelationships, such as ***Dependencies*** and ***Generalizations***, and ***Constraints***.

Different ***Models*** can be defined for the same modeled system, specifying it from different viewpoints, like a logical model, a design model, a use-case model, etc. Each ***Model*** is self-contained within its viewpoint of the modeled system and within the chosen level of abstraction.

Attributes

No extra attributes.

Associations

No extra associations.

Package

A *package* is a grouping of model elements.

In the metamodel a ***Package*** is a ***GeneralizableElement.*** A ***Package*** contains ***ModelElements*** like ***Packages***, ***Classifiers***, and ***Associations***. A ***Package*** may also contain ***Constraints*** and ***Dependencies*** between ***ModelElements*** of the ***Package***.

A ***Package*** may have «***import***» dependencies to other ***Packages***, allowing ***ModelElements*** in the other ***Packages*** to be used by ***ModelElements*** in the first ***Package***. The ***ModelElements*** available in a ***Package*** are those owned by the ***Package*** together with those referenced, i.e. owned by other, imported ***Packages***. Furthermore, each ***ModelElement*** of a ***Package*** has a *visibility* relative to the ***Package*** stating if the ***ModelElement*** is visible outside the ***Package*** or to a specialization of the ***Package***.

Attributes

No extra attributes.

Associations

referencedElement A ***Package*** references ***ModelElements*** in other, imported ***Packages***.

Subsystem

A *subsystem* is a grouping of model elements, of which some constitute a specification of the

behavior offered by the other contained model elements.

In the metamodel ***Subsystem*** is a subclass of both ***Package*** and ***Classifier,*** whose ***Features*** are all ***Operations***. The contents of a ***Subsystem*** is divided into two subsets: specification elements and realization elements. The former provides, together with the ***Operations*** of the ***Subsystem***, a specification of the behavior contained in the ***Subsystem***, while the ***ModelElements*** in the latter subset jointly provide a realization of the specification.

The specification elements are ***UseCases*** together with their offered ***Interfaces***, ***Constraints***, and relationships. The realization elements are ***Classes*** and ***Subsystems*** together with their associated ***Interfaces***, ***Constraints***, and relationships. The relationship between the specification elements and the realization elements is defined with a set of ***Collaborations***.

Attributes

isInstantiable States whether a ***Subsystem*** is instantiable or not. If true, then the instances of the model elements within the subsystem form an implicit composition to an implicit subsystem instance, whether or not it is actually implemented.

Associations

No extra associations.

12.3 WELL-FORMEDNESS RULES

The following well-formedness rules apply to the Model Management package.

ElementReference

No extra well-formedness rules.

Model

No extra well-formedness rules.

Package

[1] A ***Package*** may only own or reference ***Packages***, ***Subsystems***, ***Classifiers, Associations***, ***Generalizations***, ***Dependencies***, ***Constraints, Collaborations***, ***Messages***, and ***Stereotypes***.

```
self.contents->forAll ( c |
      c.oclIsKindOf(Package)          or
      c.oclIsKindOf(Subsystem)                or
      c.oclIsKindOf(Classifier)       or
      c.oclIsKindOf(Association)      or
      c.oclIsKindOf(Generalization)   or
      c.oclIsKindOf(Dependency)       or
      c.oclIsKindOf(Constraint)       or
      c.oclIsKindOf(Collaboration)    or
      c.oclIsKindOf(Message)          or
      c.oclIsKindOf(Stereotype) )
```

[2] No referenced element (excluding ***Association***) may have the same name or alias as any element owned by the ***Package*** or one of its supertypes.

```
self.allReferencedElements->reject( re |
```

```
re.oclIsKindOf(Association) )->forAll( re |
   (re.elementReference.alias <> '' implies
      not (self.allContents - self.allReferencedElements)->reject( ve |
         ve.oclIsKindOf (Association) )->exists ( ve |
            ve.name = re.elementReference.alias))
   and
   (re.elementReference.alias = '' implies
      not (self.allContents - self.allReferencedElements)->reject ( ve |
         ve.oclIsKindOf (Association) )->exists ( ve |
            ve.name = re.name) ) )
```

[3] Referenced elements (excluding ***Association***) may not have the same name or alias.

```
self.allReferencedElements->reject( re |
   not re.oclIsKindOf (Association) )->forAll( r1, r2 |
      (r1.elementReference.alias <> '' and r2.elementReference.alias <> '' and
         r1.elementReference.alias = r2.elementReference.alias implies r1 = r2)
      and
      (r1.elementReference.alias = '' and r2.elementReference.alias = '' and
         r1.name = r2.name implies r1 = r2)
      and
      (r1.elementReference.alias <> '' and r2.elementReference.alias = '' implies
         r1.elementReference.alias <> r2.name))
```

[4] No referenced element (***Association***) may have the same name or alias combined with the same set of associated ***Classifiers*** as any ***Association*** owned by the ***Package*** or one of its supertypes.

```
self.allReferencedElements->select( re |
   re.oclIsKindOf(Association) )->forAll( re |
      (re.elementReference.alias <> '' implies
         not (self.allContents - self.allReferencedElements)->select( ve |
            ve.oclIsKindOf(Association) )->exists( ve : Association |
               ve.name = re.elementReference.alias
               and
               ve.connection->size = re.connection->size and
               Sequence {1..re.connection->size}->forAll( i |
                  re.connection->at(i).type = ve.connection->at(i).type ) ) )
      and
      (re.elementReference.alias = '' implies
         not (self.allContents - self.allReferencedElements)->select( ve |
            not ve.oclIsKindOf(Association) )->exists( ve : Association |
               ve.name = re.name
               and
               ve.connection->size = re.connection->size and
               Sequence {1..re.connection->size}->forAll( i |
                  re.connection->at(i).type = ve.connection->at(i).type ) ) ) )
```

[5] Referenced elements (***Association***) may not have the same name or alias combined with the same set of associated ***Classifiers***.

```
self.allReferencedElements->select ( re |
   re.oclIsKindOf (Association) )->forAll ( r1, r2 : Association |
      (r1.connection->size = r2.connection->size and
      Sequence {1..r1.connection->size}->forAll ( i |
         r1.connection->at (i).type = r2.connection->at (i).type and
      r1.elementReference.alias <> '' and r2.elementReference.alias <> '' and
         r1.elementReference.alias = r2.elementReference.alias implies r1 = r2))
      and
      (r1.connection->size = r2.connection->size and
        Sequence {1..r1.connection->size}->forAll ( i |
           r1.connection->at (i).type = r2.connection->at (i).type and
         r1.elementReference.alias = '' and r2.elementReference.alias = '' and
          r1.name = r2.name implies r1 = r2))
```

```
and
(r1.connection->size = r2.connection->size and
Sequence {1..r1.connection->size}->forAll ( i |
   r1.connection->at (i).type = r2.connection->at (i).type and
r1.elementReference.alias <> '' and r2.elementReference.alias = '' implies
   r1.elementReference.alias <> r2.name)))
```

[6] The referenced elements of a ***Package*** are the public elements of imported ***Packages***, transitively.

```
self.referencedElement = self.requirement->select (d |
   d.stereotype.name = 'import').supplier.oclAsType(Package).allVisibleElements
```

[7] A ***Package*** imports all its owned ***Packages***.

```
self.requirement->select (s |
   s.stereotype.name = 'import').supplier->includesAll(
         self.ownedElement->select ( e | e.oclIsKindOf (Package) ) )
```

Additional Operations

[1] The operation *contents* results in a Set containing the ***ModelElements*** owned by or imported by the ***Package***.

```
contents : Set(ModelElement)
contents = self.ownedElement->union(self.referencedElement)
```

[2] The operation *allReferencedElements* results in a Set containing the ***ModelElements*** referenced by the ***Package*** or one of its supertypes.

```
allReferencedElements : Set(ModelElement)
allReferencedElements = self.referencedElement->union(
   self.supertype.oclAsType(Package).allReferencedElements->select( re |
      re.elementReference.visibility = #public or re.elementReference.visibility =
#protected))
```

Subsystem

[1] For each ***Operation*** in an ***Interface*** offered by a ***Subsystem***, the ***Subsystem*** itself or at least one contained ***UseCase*** must have a matching ***Operation***.

```
self.specification.allOperations->forAll(interOp |
   self.allOperations->union(self.allSpecificationElements.allOperations)->exists
      ( op | op.hasSameSignature(interOp) ) )
```

[2] The ***Features*** of a ***Subsystem*** may only be ***Operations***.

```
self.feature->forAll(f | f.oclIsKindOf(Operation))
```

[3] Each ***Operation*** must be realized by a ***Collaboration***.

```
not self.isAbstract implies self.allOperations->forAll( op |
   self.allContents->select(c |
         c.oclIsKindOf(Collaboration) )->exists(c : Collaboration|
                                                c.representedOperation = op ) )
```

[4] Each specification element must be realized by a ***Collaboration***.

```
not self.isAbstract implies self.allSpecificationElements->forAll( s |
   self.allContents->select(c |
      c.oclIsKindOf(Collaboration) )->exists(c : Collaboration|
            c.representedClassifier = s ) )
```

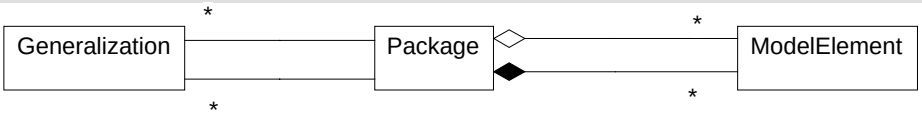
Additional Operations

[1] The operation *allSpecificationElements* results in a Set containing the ***ModelElements*** specifying the behavior of the ***Subsystem***.

```
allSpecificationElements : Set(UseCase)
allSpecificationElements = self.allContents->select(c | c.oclIsKindOf(UseCase) )
```

12.4 SEMANTICS

Package



The purpose of the *package* construct is to provide a general grouping mechanism. A package cannot be instantiated, thus it has no runtime semantics; in fact, its only semantics is to define a namespace for its contents. The package construct can be used for element organization of any purpose; the criteria to use for grouping elements together into one package are not defined within UML.

A package owns a set of model elements, with the implication that if the package is removed from the model, so are the elements owned by the package. Elements owned by the same package must have unique names within the package, although elements in different packages may have the same name.

There may be relationships between elements contained in the same package, but not a priori between an element in one package and an element outside that package. In other words, elements outside a package are by default not available to elements inside the package. There are two ways of making them available inside the package: by importing their containing packages or by defining generalizations (see below) to these other packages. An *import dependency* (a ***Dependency*** with the stereotype «***import***») from one package to another means that the first package *references* all the elements with sufficient visibility in the second package. Referenced elements are not owned by the package but they may be used in associations, generalizations, attribute types, and other relationships. A package defines the *visibility* of its contained elements to be private, protected, or public. *Private* elements are not available at all outside the containing package, *protected* elements are available only to packages with generalizations to the containing package, and *public* elements are available also to importing packages. Note, though, that the visibility mechanism does not restrict the availability of an element to peer elements in the same package.

When an element is referenced by a package it extends the namespace of that package. It is possible to give a referenced element an *alias* so it will not conflict with the names of the other elements in the namespace, including other referenced elements. The alias will be the name of that element in the namespace; the element will not appear under both the alias and its original name. If an element is not given an alias, then it must be identified using its pathname, i.e., the concatenation of the names of the enclosing packages starting with the top-most package. Furthermore, an element may have the same or a more restrictive visibility in a package referencing it than it has in the package owning it, e.g. an element that is public in one package may be protected or private to a package referencing the element.

A package importing another package references all the public contents of the namespace defined

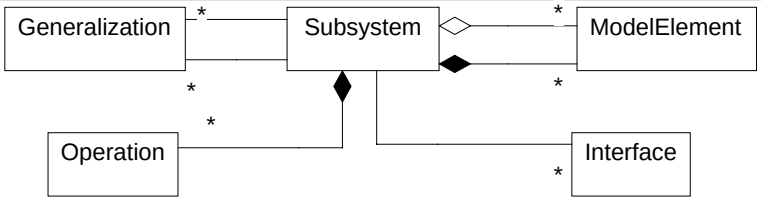
by the imported package, including elements of packages imported by the imported package. This implies that import of packages is transitive, more specifically in the following sense: Assume package A imports package B, which in turn imports package C. Then the public elements of C which are public in B are also available to A.

Packages are automatically imported by their containing package. Because of the recursiveness of import, even elements contained within several levels of packages are available, according to the visibility of contained elements. The visibility of an element contained within several levels of packages is the most restrictive of the visibilities of all containing packages.

A package can have *generalizations* to other packages. This means that the public and protected elements owned or referenced by a package are also available to its heirs, and can be used in the same way as any element referenced by the heirs themselves. Elements made available to another package by the use of a generalization appear under their real names, not under aliases. Moreover, they have the same visibility in the heir as they have in the owning package.

A package can be used to define a *framework*, consisting of patterns in the form of e.g. collaborations where (some of) the base elements are the parameters of the patterns. Apart from that, a framework package is described as an ordinary package.

Subsystem



The purpose of the *subsystem* construct is to provide a grouping mechanism with the possibility to specify the behavior of the contents. A subsystem may or may not be instantiable. A non-instantiable subsystem merely defines a namespace for its contents. The contents of a subsystem has the same semantics as that of a package, thus it consists of ownedElements and referencedElements, with unique names or aliases within the subsystem.

The contents of a subsystem is divided into two subsets: specification elements and realization elements. The specification elements are used for giving an abstract specification of the behavior offered by the realization elements.

The specification of a subsystem consists of the specification subset of the contents together with the subsystem's *features* (operations). It specifies the behavior performed jointly by instances of classifiers in the realization subset, without revealing anything about the contents of this subset. The specification is made in terms of use cases and/or operations, where use cases are used to specify complete sequences performed by the subsystem (i.e. by instances of its contents) interacting with its surroundings, while operations only specify fragments. Furthermore, the specification part of a subsystem also includes constraints, relationships between the use cases, etc.

A subsystem has no behavior of its own. All behavior defined in the specification of the subsystem is jointly offered by the elements in the realization subset of the contents. In general, since they are classifiers, subsystems can appear anywhere a classifier is expected. The general interpretation of this is that since the subsystem itself cannot be instantiated or have any behavior of its own, the requirements posed on the subsystem in the context where it occurs is fulfilled by its contents. The same is true for associations; i.e. any association connected to a subsystem is actually connected to

one of the classifiers it contains.

The correspondence between the specification part and the realization part of a subsystem is specified with a set of *collaborations*, at least one for each operation of the subsystem and for each contained use case. Each collaboration specifies how instances of the realization elements cooperate to jointly perform the behavior specified by the use case or operation, i.e. how the higher level of abstraction is transformed into the lower level of abstraction. A message instance received by an instance of a use case (higher level of abstraction) corresponds to an instance conforming to one of the classifier roles in the collaboration receiving that message instance (lower level of abstraction). This instance communicates with other instances conforming to other classifier roles in the collaboration, and together they perform the behavior specified by the use case. All message instances that can be received and sent by instances of the use cases are also received and sent by the conforming instances, although at a lower level of abstraction. Similarly, application of an operation of the subsystem actually means that a message instance is sent to a contained instance that then performs a method.

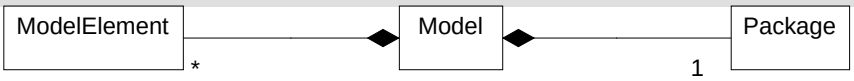
Importing of subsystems is done in the same way as for packages, using the *visibility* property to define whether elements are *public*, *protected*, or *private* to the subsystem.

A subsystem can have *generalizations* to other subsystems. This means that the public and protected elements in the contents of a subsystem are also available to its heirs. In a concrete, i.e. non-abstract, subsystem all elements in the specification, including elements from ancestors, must be completely realized by cooperating realization elements, as specified with a set of collaborations. This may not be true for abstract subsystems.

Subsystems may offer a set of *interfaces*. This means that for each operation defined in an interface, the subsystem offering the interface must have a matching operation, either as a feature of the subsystem itself or of a use case. The relationship between interface and subsystem is not necessarily one-to-one; a subsystem may realize several interfaces and one interface may be realized by more than one subsystem.

A subsystem can be used to define a *framework*, consisting of patterns in the form of e.g. collaborations where (some of) the base elements are the parameters of the patterns. Furthermore, the specification of a framework subsystem may also be parameterized.

Model



The purpose of a *model* is to describe the modeled system at a certain level of abstraction and from a specific viewpoint, such as a logical or a behavioral view of the modeled system.

A model describes the modeled system completely in the sense that it covers the whole modeled system, although only those aspects relevant within the chosen level of abstraction and viewpoint are represented in the model. The model consists of a containment hierarchy where the top-most package represents the boundary of the modeled system.

The model may also contain model elements describing relevant parts of the system's environment. The environment may be modeled by actors and their interfaces. These model elements and the model elements representing the modeled system may be associated with each other. Such associations are owned either by the model or by the top-most package. The contents of a model is the transitive closure of its owned model elements, like packages, classifiers, and relationships.

Relationships between model elements in different models have no impact on the model elements' meaning in their containing models because of the self-containment of models. Note, however, that even if inter-model relationships do not express any semantics in relation to the models, they may have semantics in relation to the reader or in deriving model elements as part of the overall development process.

A model may be a specialization of another model. This implies that all elements in the ancestor are also available in the specialized model under the same name as in the ancestor.

12.5 STANDARD ELEMENTS

The predefined stereotypes, constraints and tagged values for the Model Management package are listed in Table 6 and defined in the *Standard Elements* appendix.

Model Element	Stereotypes	Constraints	Tagged Values
Package	«facade» «framework» «stub» «system»		

Table 6: Model Management—Standard Elements

12.5 NOTES

Because this is a logical model of the UML, distribution or sharing of models between tools is not described.

The *visibility* of an element in an importing package/subsystem may be more restrictive than its visibility in the owning namespace. This is useful for example when a namespace makes parts of its contents public to the surrounding namespace, but these elements are not available to the outside of the surrounding namespace.

In UML there are three different ways to model a group of elements contained in another element: by using a *package*, a *subsystem*, or a *class*. Some pragmatics on their use:

- Packages are used when nothing but a plain grouping of elements is required.
- Subsystems provide grouping suitable for top-down development, since the requirements on the behavior of their contents can be expressed before the realization of this behavior is defined. The specification of a subsystem may also be seen as a provider of "high level APIs" of the subsystem.
- Classes are used when the container itself should be instantiable, so that it is possible to define composite objects.

APPENDICES

The following appendices provide a summary of standard elements and a glossary.

Contents

Appendix A: Standard Elements

Appendix B: Glossary

APPENDIX A: STANDARD ELEMENTS

This appendix describes the predefined standard elements for UML. The standard elements are organized into categories (stereotypes, tagged values and constraints) and are alphabetically ordered.

A.1 STEREOTYPES

The following stereotypes are predefined in the UML; any stereotype that applies to a specific class in the metamodel also applies to any subclasses of that class:

Name	Applies to	Description
«becomes»	Dependency	Becomes is a stereotyped dependency whose source and target are represent the same instance at different points in time, but each with potentially different values, state instance, and roles. A becomes dependency from A to B means that that instance A becomes B with possibly new values, state instance, and roles at a different moment in time/space.
«call»	Dependency	Call is a stereotyped dependency whose source is an operation and whose target is an operation. A call dependency specifies that the source invokes the target operation. A call dependency may connect a source operation to any target operation that is within scope, including but not limited to operations of the enclosing classifier and operations of other visible classifiers.
«copy»	Dependency	Copy is a stereotyped dependency whose source and target are different instances, but each with the same values, state instance, and roles (but a distinct identity). A copy dependency from A to B means that B is an exact copy of A. Future changes in A are not necessarily reflected in B.
«create»	BehavioralFeature	Create is a stereotyped behavioral feature denoting that the designated feature creates an instance of the classifier to which the feature is attached.
	Event	Create is a stereotyped event denoting that the instance enclosing the state machine to which the event type applies is created. Create may only be applied to an initial transition at the topmost level of this state machine, and in fact, this is the only kind of trigger that may be applied to an initial transition.

«destroy»	BehavioralFeature	Delete is a stereotyped behavioral feature denoting that the designated feature destroys an instance of the classifier to which the feature is attached.
	Event	Delete is a stereotyped event denoting that the instance enclosing the state machine to which the event type applies is destroyed.
«deletion»	Refinement	Deletion is a stereotyped refinement having no clients and no sub-refinements
«derived»	Dependency	Derived is a stereotyped dependency whose source and target are both elements, usually but not necessarily of the same type. A derived dependency specifies that the source is derived from the target, meaning that the source is not manifest, but rather is implicitly derived from the target.
«document»	Component	Document is a stereotyped component representing a document.
«enumeration»	DataType	Enumeration is a stereotyped data type, whose details specify a domain consisting of a set of identifiers that are the possible values of an instance of the data type
«executable»	Component	Executable is a stereotyped component denoting a program that may be run on a Node.
«extends»	Generalization	Extends is a stereotyped generalization between use cases. It specifies that the contents of the extending use case may be added to the related use case. It not only specifies where the contents should be added (*extensionPoint*), but also if it only should be added if a specified *condition* (BooleanExpression). When an instance of the related use case reaches the extension point and the condition is fulfilled, the instance continues according to a sequence that is the result of extending the original sequence with the extending sequence at this point. It is required that the ordering of the parts of the extending use case must be fulfilled if its parts are inserted at different places.
«facade»	Package	Facade is a stereotyped package containing nothing but references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package. A Façade does not contain any model elements of its own.
«file»	Component	File is a stereotyped component representing a document containing source code or data.

«framework»	Package	Framework is a stereotyped package consisting mainly of patterns.
«friend»	Dependency	Friend is a stereotyped usage dependency whose source is a model element, such as an operation, class, or package, (or operation) and whose target is a different package model element, such as a class or package. (or operation). A friend relationship grants the source access to the target regardless of the declared visibility. It extends the visibility of the source so that the target can see into the source.
«import»	Dependency	Import is a stereotyped dependency between two packages, denoting that the public contents of the target package are added to the namespace of the source package.
«implementationClass»	Class	Implementation class is a stereotyped class that is not a type and that represents the implementation of a class in some programming language. An instance may have zero or one implementation classes. This is in contrast to plain general classes, wherein an instance may statically have multiple classes at one time and may gain or lose classes over time and an object (a subtype of instance) may dynamically have multiple classes.
«inherits»	Generalization	Inherits is a stereotyped generalization denoting that instances of the subtype are not substitutable for instance of the supertype.
«instance»	Dependency	Instance is a stereotyped dependency whose source is an instance and whose target is a classifier. An instance dependency from I to C means that I is an instance of C.
«invariant»	Constraint	Invariant is a stereotyped constraint that must be attached to a set of classifiers or relationships, and denotes that the conditions of the constraint must hold for the classifiers or relationships and their instances.
«library»	Component	Library is a stereotyped component representing a static or dynamic library.
«metaclass»	Dependency	Metaclass is a stereotyped dependency whose source and target are both classifiers and denoting that the target is the metaclass of the source.
	Classifer	Metaclass is a stereotyped classifier denoting that the class is a metaclass of some other class.
«postcondition»	Constraint	Postcondition is a stereotyped constraint that must be attached to an operation, and denotes that the conditions

		of the constraint must hold after the invocation of the operation.
«powertype»	Classifier	Powertype is a stereotyped classifier denoting that the classifier is a metatype, whose instances are subtypes of another type.
	Dependency	Powertype is a stereotyped dependency whose source is a set of generalizations and whose target is a classifier specifying that the target is the powertype of the source.
«precondition»	Constraint	Precondition is a stereotyped constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold for the invocation of the operation.
«private»	Generalization	Private is a stereotyped generalization of that specifies private inheritance. It hides the inherited features of a class and therefore renders it non-substitutable for declarations of its ancestors.
«process»	Classifier	Process is a stereotyped classifier that is also an active class, representing a heavy-weight flow of control.
«requirement»	Comment	Requirement is a stereotyped comment that states a responsibility or obligation.
«send»	Dependency	Send is a stereotyped dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.
«stereotype»	Classifier	Stereotype is a stereotyped classifier, denoting that the classifier serves as a stereotype. This stereotype permits modelers to model stereotype hierarchies.
«stub»	Package	Stub is a stereotyped package representing a package that is incomplete transferred; specifically, a stub provides the public parts of the package, but nothing more.
«subclass»	Generalization	Subclass is a stereotyped generalization denoting that instances of the subtype are not substitutable for instance of the supertype
«subtraction»	Refinement	Subtraction is a stereotyped refinement having no clients and no sub-refinements
«subtype»	Generalization	Subtype is a stereotyped generalization that offers no different properties or behavior than basic generalization. This stereotype exists as the opposite of

		subclass, so that subtyping versus subclassing can be explicitly marked.
«system»	Package	System is a stereotyped package that represents a collection of models of the same modeled system. The models contained in the System all describe the modeled system from different viewpoints, the viewpoints not necessarily disjoint. The System therefore makes up a comprehensive specification of the modeled system—it is the top-most construct in the specification. A System also contains all relationships and constraints between model elements contained in different models. These model elements add no semantic information to the connected model elements, since each model shows a complete view of the modeled system. Thus, these model elements do not express information on the modeled system as such, but rather on the models. e.g., they may be used for requirements tracking. A modeled system may be realized by a set of *subordinate* modeled systems, each described by its own set of models collected in a separate System. A System can only be contained in a System.
«table»	Component	Table is a stereotyped component representing a data base table.
«thread»	Classifier	Thread is a stereotyped classifier that is also an active class, representing a light-weight flow of control.
«topLevelPackage»	Package	TopLevelPackage is a stereotyped package denoting the top-most package in a model, representing all the non-environmental parts of the model. A TopLevelPackage is at the top of the containment hierarchy in a model.
«type»	Class	Type is a stereotype of Class**,** meaning that the class is used for specification of a domain of instances (objects) together with the operations applicable to the objects. A type may not contain any methods but it may have attributes and associations.
«useCaseModel»	Model	UseCaseModel is a model that describes a system's functional requirements in terms of a set of use cases and their interactions with actors. It is required that a UseCaseModel only contains use cases and actors and their relationships: *extends* and *uses* between use cases, associations between use cases and actors, and generalizations between actors.
«uses»	Generalization	Uses is a stereotyped generalization between use cases. It specifies that the contents of the related use case is

		included (or used) in the description of the other use case. It is typically used for extracting shared behavior. It requires that the ordering of the parts of the used use case must be fulfilled if its parts are used at different places. Uses may only be defined between use cases.
«utility»	Classifier	Utility is a stereotyped classifier representing a classifier that has not instances but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped

A.2 TAGGED VALUES

The following tagged values are predefined in the UML; any tagged value that applies to a specific class in the metamodel also applies to any subclasses of that class:

Name	Applies to	Description
documentation	Element	Documentation is a comment, description, or explanation of the element to which it is attached.
location	Classifier	Location denotes that the classifier is a part of the given component.
	Component	Location denotes that the component resides on given node.
persistence	Attribute	Persistence denotes the permanence of the state of the attribute, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
	Classifier	Persistence denotes the permanence of the state of the classifier, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
	Instance	Persistence denotes the permanence of the state of the instanced, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
responsibility	Classifier	Responsibility is a contract by or an obligation of the classifier.
semantics	Classifier	Semantics is the specification of the meaning of the classifier.

	Operation	Semantics is the specification of the meaning of the operation.

A.3 CONSTRAINTS

The following constraints are predefined in the UML:

Name	Applies to	Description
association	LinkEnd	Association is a constraint applied to a link-end, specifying that the corresponding instance is visible via association
broadcast	Request	Broadcast is a constraint applied to a request sent to multiple instances, specifying that it is sent simultaneously to all target instances, in an undefined unspecified order.
complete	Generalization	Complete is a constraint applied to a set of generalizations, specifying that all subtypes have been specified (although some may be elided) and that not additional subtypes are permitted.
disjoint	Generalization	Disjoint is a constraint applied to a set of generalizations, specifying that instance may have no more than one of the given subtypes as a type of the instance. This is the default semantics of generalization.
global	LinkEnd	Global is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is in a global scope relative to the link.
implicit	Association	Implicit is a constraint applied to an association, specifying that the association is not manifest, but rather is only conceptual.
incomplete	Generalization	Incomplete is a constraint applied to a set of generalizations, specifying that not all subtypes have been specified (even if some are elided) and that additional subtypes are permitted. This is the default semantics of generalizations.
local	LinkEnd	Local is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is in a local scope relative to the link.
or	Association	Or is a constraint applied to a set of associations, specifying that over that set, only one is manifest for

		each associated instance. Or is an exclusive (not inclusive) constraint.
overlapping	Generalization	Overlapping is a constraint applied to a set of generalizations, specifying that instances may have more than one of the given subtypes as a type of the instance.
parameter	LinkEnd	Parameter is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is a parameter relative to the link
self	LinkEnd	Self is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is the dispatcher of a request.
vote	Request	Vote is a constraint applied to a request, specifying that the return value is selected by a majority vote of all the return values returned from multiple instances.

APPENDIX B: GLOSSARY

This glossary defines the terms that are used to describe the Unified Modeling Language (UML). In addition to UML-specific terminology it includes related terms from OMG standards and object-oriented analysis and design methods. The glossary is intended for use by anyone who wants to understand UML concepts and background. Glossary entries are organized alphabetically.

B.1 SCOPE

This glossary includes terms from the following primary sources:

- UML Semantics

UML Notation Guide In addition the following secondary sources have been used:

- Object Management Architecture object model [OMA]
- CORBA 2.0 [CORBA]
- Object Analysis & Design RFP-1 [OA&D RFP]

[OMA], [CORBA] and [OA&D RFP] have been used to promote OMG-compliancy and provide distributed object terms that complement UML. (When there are inconsistencies among the three OMG sources, we have ranked their authority in the order listed.)

B.2 NOTATION CONVENTIONS

The entries in the glossary usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.

When one or more words in a multi-word term is enclosed by brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.

The following conventions are used in this glossary:

- Contrast: <term>. Refers to a term that has an opposed or substantively different meaning.
- See: <term>. Refers to a related term that has a similar, but not synonymous meaning.
- Synonym: <term>. Indicates that the term has the same meaning as another term, which is referenced.
- Acronym: <term>. This indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.

abstract class
A class that cannot be directly instantiated. Contrast: *concrete class*.

abstraction
The essential characteristics of an entity that distinguish it from all other kind of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action
The specification of an executable statement that forms an abstraction of a computational procedure. An action results in a change in the state of the model, and is realized by sending a message to an object or modifying a value of an attribute.

action expression
An expression that resolves to a collection of actions.

action state
A state that represents the execution of an atomic action, typically the invocation of an operation.

activation
The execution of an action. Contrast: *activation* [OMA].

active class
A class whose instances are active objects. See: *active object*.

active object
An object that owns a thread and can initiate control activity. An instance of active class. See: *active class*, *thread*.

activity diagram
A special case of a state diagram in which all or most of the states are action states and in which all or most of the of the transitions are triggered by completion of actions in the source states. Contrast: *state diagram*.

actor [class]
A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

actual parameter
Synonym: *argument*.

aggregate [class]
A class that represents the “whole” in an aggregation (whole-part) relationship. See: aggregation.

aggregation
A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. Contrast: *composition*.

analysis
The part of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses what to do, design focuses on how to do it. Contrast: *design*.

analysis time
Refers to something that occurs during an analysis phase of the software development process. See: *design time*, *modeling time*.

architecture
The organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts.

argument
A specific value corresponding to a parameter. Synonym: actual parameter. Contrast: *parameter*.

artifact
A piece of information that is used or produced by a software development process. An artifact can be a model, a description or software.

association
The semantic relationship between two or more classifiers that involves connections among their instances.

association class
A modeling element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end
The endpoint of an association, which connects the association to a classifier.

asynchronous action
A request where the sending object does not pause to wait for results. Synonym: *asynchronous request* [OMA]. Contrast*: synchronous action*.

attribute
A named slot within a classifier that describes a range of values that instances of the classifier may hold. Synonym: *attribute* [OMA].

behavior
The observable effects of an operation or event, including its results. Synonym: *behavior* [OMA].

behavioral feature
A dynamic feature of a model element, such as an operation or method.

behavioral model aspect
A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories

binary association
An association between two classes. A special case of an n-ary association.

binding
The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean
An enumeration whose values are true and false.

boolean expression
An expression that evaluates to a boolean value.

cardinality
The number of elements in a set. Contrast: multiplicity.

class
A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. Synonym: *class* [OMA]. See: *interface*.

classifier
A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes and datatypes and components.

class diagram
A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client
A classifier that requests a service from another classifier. Synonym: *client object* [OMA]. Contrast: *supplier*.

collaboration
The specification of how a classifier, such as a use case or operation, is realized by a set of classifiers and associations playing specific roles. used in a specific way. The collaboration defines an interaction. See: *interaction*.

collaboration diagram
A diagram that shows interactions organized around instances and their links to each other. Unlike a sequence diagram a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *sequence diagram*.

comment
An annotation attached to an element or a collection of elements. A note has no semantics. Contrast: *constraint*.

communication association
In a deployment diagram an association between nodes that implies a communication. See: *deployment diagram*.

compile time
Refers to something that occurs during the compilation of a software module. See: *modeling time*, *run time*.

component
An executable software module with identity and a well-defined interface. Contrast: *component* [OMA].

component diagram
A diagram that shows the organizations and dependencies among components.

composite [class]
A class that is related to one or more classes by a composition relationship. See: composition.

composite aggregation
Synonym: *composition*.

composite state
A state that consists of either concurrent substates or disjoint substates. Contrast: substate.

composition
A form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive. Synonym: *composite aggregation*.

concrete class
A class that can be directly instantiated. Contrast: *abstract class*.

concurrency
The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See: *thread*.

concurrent substate
A substate that can be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *disjoint substate*.

constraint
A semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. Constraints are one of three extendibility mechanisms in UML. See: *tagged value, stereotype.*

container
1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. For example, arrays, lists, sets. 2. A component that exists to contain other components.

containment hierarchy
A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms an acyclic graph.

context
A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

datatype
A type whose values have no identity. Datatypes include primitive built-in types (such as numbers and strings) as well as enumeration types (such as boolean).

delegation
The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Synonym: *delegation* [OMA]. Contrast: *inheritance*.

dependency
A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

deployment diagram
A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See: *component diagrams.*

derived element
A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design
The part of the software development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time
Refers to something that occurs during a design phase of the software development process. See: *modeling time*. Contrast: *analysis time*.

development process
A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram
A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the following diagrams: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, component diagram, and deployment diagram.

disjoint substate
A substate that cannot be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *concurrent substate*.

distribution unit
A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

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

dynamic classification
A semantic variation of generalization in which an object may change type or role. Contrast: *static classification*.

element
An atomic constituent of a model.

enumeration
A list of named values used as the range of a particular attribute type. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with the values {false, true}.

event
The specification of a significant occurrence that has a location in time and space. In the context of state diagrams, an event is an occurrence that can trigger a state transition.

export
In the context of packages, to make an element visible outside its enclosing namespace. See: *visibility*. Contrast: *export* [OMA], *import*.

expression
A string that evaluates to a value of a particular type. For example, the expression "(7 + 5 * 3)" evaluates to a value of type number.

extends
A relationship from one use case to another, specifying how the behavior defined for the first use case can be inserted into the behavior defined for the second use case.

feature
A property, like operation or attribute, which is encapsulated within another entity, such as an interface, a class or a datatype.

fire
To execute a state transition. See: *transition*.

focus of control
A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

formal parameter
Synonym: *parameter*.

framework
A micro-architecture that provides an extensible template for applications within a specific domain.

generalizable element
A model element that may participate in a generalization relationship. See: *generalization*.

generalization
A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. Synonym: *generalization*

[OMA]. See: *inheritance*.

guard condition
A condition that must be satisfied in order to enable an associated transition to fire.

implementation
A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation. Synonym: *implementation* [OMA].

implementation inheritance
The inheritance of the implementation of a more specific element. Includes inheritance of the interface. Synonym: *implementation inheritance*. Contrast: *interface inheritance*.

import
In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *import* [OMA], *export*.

inheritance
The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See *generalization*. Synonym: *inheritance* [OMA].

instance
An entity to which a set of operations can be applied and which has a state that stores the effects of the operations. See: *object*.

interaction
A specification of how messages are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See *collaboration*.

interaction diagram
A generic term that applies to several types of diagrams that emphasize object interactions. These include: collaboration diagrams, sequence diagrams, and activity diagrams

interface
A declaration of a collection of operations that may be used for defining a service offered by an instance. Synonym: *interface* [OMA].

interface inheritance
The inheritance of the interface of a more specific element. Does not includes inheritance of the implementation. Synonym: *interface inheritance* [OMA]. Contrast: *implementation inheritance*.

layer
A specific way of grouping packages in a model at the same level of abstraction.

link
A semantic connection among a tuple of objects. An instance of an association. Synonym: *link* [OMA]. See: *association*.

link end
An instance of an association end. See: *association end*.

message
A specification of a communication between instances that conveys information with the expectation that activity will ensue. The receipt of a message instance is normally considered an instance of an event.

metaclass
A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel
A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel
A model that defines the language for expressing a model. An instance of a meta-metamodel.

metaobject
A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations. Synonym: *metaobject* [OMA].

method
The implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation. Synonym: *method* [OMA].

model
A semantically closed abstraction of a system. See: *system*.

model aspect
A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model element
An element that is an abstraction drawn from the system being modeled. Contrast: *view element*.

modeling time
Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note: When discussing object systems it is often important to distinguish between modeling-time and run-time concerns. See: *analysis time*, *design time*. Contrast: *run time*.

module
A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See: *component*.

multiple classification
A semantic variation of generalization in which an object may belong directly to more than one class. See: *dynamic classification*.

multiple inheritance
A semantic variation of generalization in which a type may have more than one supertype. Contrast: *single inheritance*.

multiplicity
A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: *cardinality*.

n-ary association
An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast: *binary association*.

name
A string used to identify a model element.

namespace
A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: *name*.

node
A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Run-time objects and components may reside on nodes.

object
An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations, methods, and state machines. An object is an instance of a class. Synonym: *object* [OMA]. See: *class*, *instance*.

object diagram
A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a collaboration diagram. See: *class diagram*, *collaboration diagram*.

object lifeline
A line in a sequence diagram that represents the existence of an object over a period of time. See: *sequence diagram*.

operation
A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible. Synonym: *operation* [OMA].

package
A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages. A system may be thought of as a single high-level package, with everything else in the system contained in it.

parameter
The specification of a variable that can be changed, passed or returned. A parameter may include a name, type and direction. Parameters are used for operations, messages and events. Synonyms: *parameter* [OMA], *formal parameter*. Contrast: *argument*.

parameterized element
The descriptor for a class with one or more unbound parameters. Synonym: *template*.

participates
A relationship that indicates the role that an instance plays in a modeling element. For example, a class participates in an association, an actor participates in a use case. Contrast: *participate* [OMA].

persistent object
An object that exists after the process or thread that created it has ceased to exist. Synonym: *persistent object* [OMA].

postcondition
A constraint that must be true at the completion of an operation.

precondition
A constraint that must be true when an operation is invoked.

primitive type
A predefined basic type, such as an integer or a string.

process
A thread that can execute concurrently with other threads.

product
The artifacts of development, such as models, code, documentation, work plans.

projection
A mapping from a set to a subset of it.

property
A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See: *tagged value*. Synonym: *property* [OMA].

pseudo-state
A vertex in a state machine that has the form of a state, but doesn't behave as a state. Pseudo-states include initial, final, and history vertices.

qualifier
An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association. **reception**
A declaration that a classifier is prepared to react to the receipt of a signal.

receive [a message]
The handling of a message instance passed from a sender object. See: *sender*, *receiver*.

receiver [object]
The object handling a message instance passed from a sender object. Contrast: *sender*.

reference
1. A denotation of a model element. 2. A named slot within a classifier that facilitates navigation to other classifiers.

refinement
A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relationship
A semantic connection among model elements. Examples of relationships include associations and generalizations.

repository
A storage place for object models, interfaces and implementations.

request
A request is the specification of a stimulus being sent to instances. It can be either an operation or a signal.

requirement
A desired feature, property, or behavior of a system.

responsibility
A contract or obligation of a type or class.

reuse
The use of a pre-existing artifact.

role
The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association end) or dynamic (e.g., a collaboration role).

run time
The period of time during which a computer program executes. Contrast: *modeling time*.

scenario
A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction. See:

interaction.

semantic variation point
A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send [a message]
The passing of a message instance from a sender object to a receiver object. See: *sender*, *receiver*.

sender [object]
The object passing a message instance to a receiver object. Contrast: *receiver*.

sequence diagram
A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See: *collaboration diagram*.

signal
The specification of an asynchronous stimulus communicated between instances. Signals may have parameters.

signature
The name and parameters of a behavioral feature. A signature may include an optional returned parameter. Synonym: *signature* [OMA].

single inheritance
A semantic variation of generalization in which a type may have only one supertype. Synonym: *multiple inheritance* [OMA]. Contrast: *multiple inheritance*.

specification
A declarative description of what something is or does. Contrast: *implementation*.

state
A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: *state* [OMA].

statechart diagram
A diagram that shows a state machine. See: *state machine*.

state machine
A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

static classification
A semantic variation of generalization in which an object may not change type or may not change role. Contrast: *dynamic classification*.

stereotype
A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extendibility mechanisms in UML. See: *constraint*, *tagged value*.

string
A sequence of text characters. The details of string representation depend on implementation, and may include character sets that support international characters and graphics.

structural feature
A static feature of a model element, such as an attribute.

structural model aspect
A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes and operations.

subclass
In a generalization relationship the specialization of another class, the superclass. See: *generalization*. Contrast: *superclass*.

substate
A state that is part of a composite state. See: *concurrent state*, *disjoint state*.

subsystem
A subsystem is a grouping of model elements, of which some constitute a specification of the behavior offered by the other contained model elements. See *package*. Contrast: *system*.

subtype
In a generalization relationship the specialization of another type, the supertype. See: *generalization*. Contrast: *supertype*.

superclass
In a generalization relationship the generalization of another class, the subclass. See: *generalization*. Contrast: *subclass*.

supertype
In a generalization relationship the generalization of another type, the subtype. Synonym: *supertype* [OMA]. See: *generalization*. Contrast: *subtype*.

supplier
A type, class or component that provides services that can be invoked by others. Synonym: *server object* [OMA]. Contrast: *client*.

swimlane
A partition on interaction diagrams for organizing responsibilities for actions. They often correspond to organizational units in a business model.

synchronous action
A request where the sending object pauses to wait for results. Synonym: *synchronous request* [OMA]. Contrast: *asynchronous action*.

system
A collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints.

tagged value
The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extendibility mechanisms in UML. See: *constraint*, *stereotype*.

template
Synonym: *parameterized element*.

thread [of control]
A single path of execution through a program, a dynamic model, or some other representation of control flow. See *process*.

time
A value representing an absolute or relative moment in time.

time event
An event that denotes the time elapsed since the current state was entered. See: *event*.

time expression
An expression that resolves to an absolute or relative value of time.

timing mark
A denotation for the time at which an event or message occurs. Timing marks may be used in constraints.

trace
A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

transient object
An object that exists only during the execution of the process or thread that created it. Synonym: *transient object* [OMA].

transition
A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state the transition is said to fire.

type
A stereotype of class that is used to specify a domain of instances (objects) together with the operations applicable to the objects. A type may not contain any methods. Synonym: *type* [OMA]. See: *class*, *instance*. Contrast: *interface*.

type expression
An expression that evaluates to a reference to one or more types.

uninterpreted
A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See: *any* [CORBA].

usage
A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

use case [class]
The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See: *use case instances*.

use case diagram
A diagram that shows the relationships among actors and use cases within a system.

use case instance
The performance of a sequence of actions being specified in a use case. An instance of a use case. See: *use case class*.

use case model
A model that describes a system's functional requirements in terms of use cases.

uses
A relationship from a use case to another use case in which the behavior defined for the former use case employs the behavior defined for the latter.

utility
A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct but a programming convenience.

value
An element of a type domain. Contrast: *value* [OMA].

vertex
A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See: *state*, *pseudo-state*.

view
A projection of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

view element
A view element is a textual and/or graphical projection of a collection of model elements.

view projection
A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

visibility
An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

INDEX

A

Action, **74**, **84**
ActionSequence, **74**
ActionState, **128**, **130**, **131**
ActivityModel, **127**, **130**, **131**
ActivityState, **128**
Actor, **96**, 97, 98
AggregationKind, **66**
Argument, **74**
Association, **23**, 33, 43
AssociationClass, **24**, 34, 45
AssociationEnd, **24**, 34
AssociationEndRole, **87**, **90**
AssociationRole, **88**, **90**
Attribute, **25**, 34
AttributeLink, **75**, **79**

B

BehavioralFeature, **26**, 34
Binding, **50**, 53
Boolean, **67**
BooleanExpression, **67**

C

CallAction, **75**, 79
CallEvent, **105**
ChangeableKind, **67**
ChangeEvent, **105**, 109
Class, **26**, 35, 41
Classifier, **27**, 35
ClassifierInState, **129**
ClassifierRole, **88**, **90**
Collaboration, **88**, **90**
Collaboration, 92
Comment, **50**, 54
Component, **51**, 54
CompositeState, **105**, 109
Constraint, **27**, 37, **59**, 62
CreateAction, **75**, 80

D

DataType, **28**, 37
DataValue, **76**, **80**
Dependency, **28**, 37, **51**, 54, 147
DestroyAction, **75**, 80

E

Element, **28**, 37
ElementReference, **135**, 137
Enumeration, **67**
EnumerationLiteral, **67**
Event, **106**
Exception, **76**
Expression, **67**

F

Feature, **29**, 37

G

GeneralizableElement, **29**, 37
Generalization, **30**, 38
Geometry, **67**
GraphicMarker, **67**
Guard, **106**, 110

I

Instance, **76**, **80**, 147
instance of, 147
Integer, **67**
Interaction, **89**, **91**, 93
Interface, **30**, 38, 43

L

Link, **83**
Link, **76**, **81**
LinkEnd, **77**, **81**
LinkObject, **77**, **81**
LocalInvocation, **77**, 110

M

Mapping, **67**
Message, **89**, **91**
MessageDirectionKind, **67**
MessageInstance, **77**, **81**
Method, **31**, 38
Model, **136**, 137, 142
ModelElement, **31**, 38, **51**, 54, 62
Multiplicity, **67**
MultiplicityRange, **68**

N

Name, **68**
namespace, 27
Namespace, **31**, 39
Node, **52**, 54

O

Object, **78**, **82**
Object and DataValue, **83**
ObjectFlowState, **129**, **130**, **131**
ObjectSetExpression, **68**
Operation, **32**, 40
OperationDirectionKind, **68**
OwnedElement, **28**, 37

P

Package, **136**, 137, 140
Parameter, **32**, 40
ParameterDirectionKind, **68**
Partition, **129**
Presentation, **52**, 54
PrimitiveType, **68**
ProcedureExpression, **68**
PseudoState, **106**, 110, **129**, **130**
PseudostateKind, **68**

R

RaiseAction, **78**, 82
Reception, **78**, **82**
Refinement, **52**, 54
Request, **78**, **83**

S

ScopeKind, **68**
Signal, **79**, **82**
SignalAction, **107**
source, 147
State, **107**
StateMachine, **107**, 110
StateVertex, **108**, **118**
Stereotype, **60**, 62
String, **68**
StructuralFeature, **33**, 40
Structure, **68**
SubmachineState, **108**
Subsystem, **136**, 139, 141
SynchronousKind, **69**

T

TaggedValue, **61**, 63
target, 147
Template, 55
TerminateAction, **79**, 82
Time, **69**
TimeEvent, **109**
TimeExpression, **69**
Trace, **53**, 55
Transition, **109**, 111

U

Uninterpreted, **69**
UninterpretedAction, **79**
Usage, **53**, 55
UseCase, **96**, 97, 98, 99
UseCaseInstance, **97**, 98

V

ViewElement, **53**, 55
VisibilityKind, **69**