OMG Unified Modeling Language Specification

A

Version 1.1, March 1998

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form at *http://www.omg.org/library/issuerpt.htm*.

Preface

0.1 About the Unified Modeling Language (UML)

The Unified Modeling Language (UML) provides system architects working on Object Analysis and Design with one consistent language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling.

This specification represents the state-of-the-art convergence of best practices in the object-technology industry. UML is the proper successor to the object modeling languages of three previously leading object-oriented methods (Booch, OMT, and OOSE). The UML is the union of these modeling languages and more, since it includes additional expressiveness to handle modeling problems that these methods did not fully address.

One of the primary goals of UML is to advance the state of the industry by enabling OO visual modeling tool interoperability. However, in order to enable meaningful exchange of model information between tools, agreement on semantics and notation is required. UML meets the following requirements:

- Formal definition of a common OA&D meta-model to represent the semantics of OA&D models, which include static models, behavioral models, usage models, and architectural models.
- IDL specifications for mechanisms for model interchange between OA&D tools. This document includes a set of IDL interfaces that support dynamic construction and traversal of a user model.
- A human-readable notation for representing OA&D models. This document defines the UML notation, an elegant graphic syntax for consistently expressing the UML's rich semantics. Notation is an essential part of OA&D modeling and the UML.

0.2 About the Object Management Group (OMG)

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization’s charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

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

Contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820 4300
Fax: +1-508-820 4303
pubs@omg.org
http://www.omg.org

OMG’s adoption of the UML specification reduces the degree of confusion within the industry surrounding modeling languages. It settles unproductive arguments about method notations and model interchange mechanisms and allows the industry to focus on higher leverage, more productive activities. Additionally, it enables semantic interchange between visual modeling tools.

0.3 About This Document

This document is intended primarily as a precise and self-consistent definition of the UML’s semantics and notation. The primary audience of this document consists of the Object Management Group, standards organizations, book authors, trainers, and tool builders. The authors assume familiarity with object-oriented analysis and design methods. The document is not written as an introductory text on building object models for complex systems, although it could be used in conjunction with other materials or instruction. The document will become more approachable to a broader audience as additional books, training courses, and tools that apply to UML become available.

The Unified Modeling Language specification defines compliance to the UML, covers the architectural alignment with other technologies, and is comprised of the following topics:

UML Summary (Chapter 1) - provides an introduction to the UML, discussing motivation and history.

UML Semantics (Chapter 2) - defines the right semantics of the Unified Modeling Language. The UML is layered architecturally and organized by package. Within each package, the model elements are defined in the following terms:

1. Abstract syntax	UML class diagrams are used to present the UML metamodel, its concepts (metaclasses), relationships, and constraints. Definitions of the concepts are included.
2. Well-formedness rules	The rules and constraints on valid models are defined. The rules are expressed English prose and in a precise Object Constraint Language (OCL). OCL is a specification language that uses simple logic for specifying invariant properties of systems comprising sets and relationships between sets.
3. Semantics	The semantics of model usage are described in English prose.

UML Notation Guide (Chapter 3) - represents the graphic syntax for expressing the semantics described by the UML metamodel. Consequently, the UML Notation Guide's chapters should be read in conjunction with the UML Semantics chapters.

UML Extensions (Chapter 4) - contains the UML Extension for Objectory Process for Software Engineering and UML Extension for Business Modeling.

OA&D CORBAfacility Interface Definition (Chapter 5) - contains the UML-consistent interoperability defined in terms of CORBA IDL.

In addition, you will find an appendix of Standard Elements and an appendix that contains the Object Contraint Language (OCL) syntax, semantics, and grammar. All OCL features are described in terms of concepts from the UML Semantics chapter.

0.3.1 Dependencies Between Sections

UML Semantics (Chapter 2) can stand on its own, relative to the others, with the exception of the *OCL Specification*. The semantics and the OCL are interdependent. The semantics and notation are nearly independent. What this means is that you can certainly specify and understand each one in isolation, but the one affects the other. For example, knowing what kinds of things a developer or modeler finds important to visualize impacts what kind of underlying semantics are needed. For example, modeling patterns is something that in our experience we find to be valuable for systems of scale; this is why the UML metamodel has collaborations as a first-class citizen. If one does not consider what is important to be visualized, you end up with a less rich metamodel.

The *UML Notation Guide* and *OA&D CORBAfacility Interface Definition* both depend on the semantics. We consider it advantageous to separate the UML definition and the facility interface. Having these as separate standards will permit their evolution in the most flexible way, even though they are not completely independent.

The specifications in the *UML Extension* documents depend on both the notation and semantics sections.

0.4 Compliance to the UML

The UML and corresponding facility interface definition are comprehensive. However, these specifications are packaged so that subsets of the UML and facility can be implemented without breaking the integrity of the language. The UML Semantics is packaged as follows:

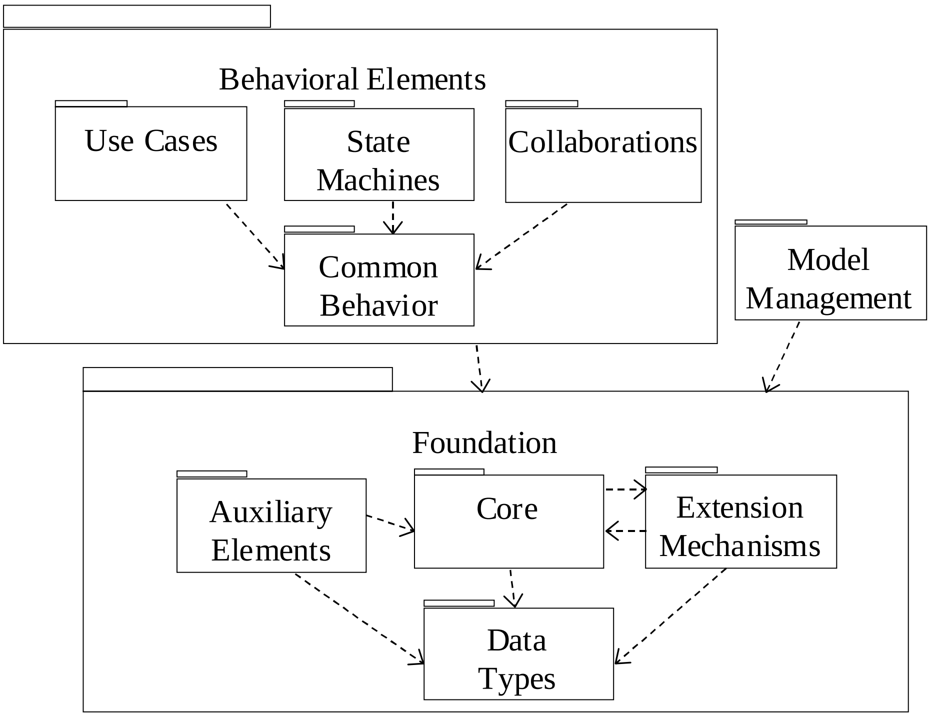


Figure 0-1 UML Class Diagram Showing Package Structure

This packaging shows the semantic dependencies between the UML model elements in the different packages. The IDL in the facility is packaged almost identically. The notation is also “packaged” along the lines of diagram type. Compliance of the UML is thus defined along the lines of semantics, notation, and IDL.

Even if the compliance points are decomposed into more fundamental units, vendors implementing UML may choose not to fully implement this packaging of definitions, while still faithfully implementing some of the UML definitions. However, vendors who want to precisely declare their compliance to UML should refer to the precise language defined herein, and not loosely say they are "UML compliant."

0.4.1 Compliance to the UML Semantics

The basic units of compliance are the packages defined in the UML metamodel. The full metamodel includes the corresponding semantic rigor defined in the Semantics section.

The class diagram illustrates the package dependencies, which are also summarized in the table below.

Table 0-1 Metamodel Packages

Package	**Prerequisite Packages**
DataTypes	
Core	DataTypes
Auxiliary Elements	Core, DataTypes
Common Behavior	Core, DataTypes
State Machines	Common Behavior, Core, DataTypes
Collaboration	Common Behavior, Core, DataTypes
Use Cases	Collaboration, Common Behavior, Core, DataTypes
Model Management	Core, DataTypes
Extension Mechanisms	Core, DataTypes

Complying with a package requires complying with the prerequisite package.

The semantics are defined in an implementation language-independent way. An implementation of the semantics, without consistent interface and implementation choices, does not guarantee tool interoperability. See the *OA&D CORBAfacility Interface Definition* (chapter 16).

In addition to the above packages, compliance to a given metamodel package must load or know about the predefined UML standard elements (i.e., values for all predefined stereotypes, tags, and constraints). These are defined throughout the semantics and notation documents and summarized in the UML Standard Elements appendix. The predefined constraint values must be enforced consistent with their definitions. Having tools know about the standard elements is necessary for the full language and to avoid the definition of user-defined elements that conflict with the standard UML elements. Compliance to the UML Extensions is defined separate from the UML Semantics, so not all tools need to know about the UML Extensions a priori.

For any implementation of UML, it is optional that the tool implement the Object Constraint Language. A vendor conforming to OCL support must support the following:

- Validate and store syntactically correct OCL expressions as values for the UML data types BooleanExpression, Expression, ObjectSetExpression, TimeExpression, and ProcedureExpression.
- Be able to perform a full type check on the object constraint expression. This check will test whether all features used in the expression are actually defined in the UML model and used correctly.

All tools conforming to the UML semantics are expected to conform to the following aspects of the semantics:

- its abstract syntax (i.e., the concepts, valid relationships, and constraints expressed in the corresponding class diagrams),
- well-formedness rules, and
- semantics.

However, vendors are expected to apply some discretion on how strictly the well-formedness rules are enforced; tools should be able to report on well-formedness violations, but not necessarily force all models to be well formed. Incomplete models are common during certain phases of the development lifecycle, so they should be permitted. See the *OA&D CORBAfacility Interface Definition* (chapter16) for its treatment of well-formedness exception handling, as an example of a technique to report well-formedness violations.

0.4.2 Compliance to the UML Notation

The UML notation is an essential element of the UML to enable communication between team members. Compliance to the notation is optional, but the semantics are not very meaningful without a consistent way of expressing them.

Notation compliance is defined along the lines of the UML Diagrams types: use case, class, statechart, activity, sequence, collaboration, component, and deployment diagrams.

If the notation is implemented, a tool must enforce the underlying semantics and maintain consistency between diagrams if the diagrams share the same underlying model. By this definition, a simple "drawing tool" cannot be compliant to the UML notation.

There are many optional notation adornments. For example, a richly adorned class icon may include an embedded stereotype icon, a list of properties (tagged values and metamodel attributes), constraint expressions, attributes with visibilities indicated, and operations with full signatures. Complying with class diagram support implies the ability to support all of the associated adornments.

Compliance to the notation in the *UML Extensions* is described separately.

0.4.3 Compliance to the UML Extensions

Vendors should specify whether they support each of the UML Extensions or not. Compliance to an extension means knowledge and enforcement of the semantics and corresponding notation.

0.4.4 Compliance to the OA&D CORBAfacility Interface Definitions

The IDL modules defined in the OA&D CORBAfacility parallel the packages in the semantic metamodel. The exception to this is that DataTypes and Extension mechanisms have been merged in with the core for the facility. Except for this, a CORBAfacility implementing the interface modules have the same compliance point options as described in “Compliance to the UML Notation” listed above.

0.4.5 Summary of Compliance Points

Table 0-2 Summary of Compliance Points

Compliance Point	Valid Options
Core	no/incomplete, complete, complete including IDL
Auxiliary Elements	no/incomplete, complete, complete including IDL
Common Behavior	no/incomplete, complete, complete including IDL
State Machines	no/incomplete, complete, complete including IDL
Collaboration	no/incomplete, complete, complete including IDL
Use Cases	no/incomplete, complete, complete including IDL
Model Management	no/incomplete, complete, complete including IDL
Extension Mechanisms	no/incomplete, complete, complete including IDL
OCL	no/incomplete, complete
Use Case diagram	no/incomplete, complete
Class diagram	no/incomplete, complete
Statechart diagram	no/incomplete, complete
Activity diagram	no/incomplete, complete
Sequence diagram	no/incomplete, complete
Collaboration diagram	no/incomplete, complete
Component diagram	no/incomplete, complete
Deployment diagram	no/incomplete, complete
UML Extension: Business Engineering	no/incomplete, complete
UML Extension: Objectory Process for Software Engineering	no/incomplete, complete

0.5 Acknowledgements

The UML was crafted through the dedicated efforts of individuals and companies who find UML strategic to their future. This section acknowledges the efforts of these individuals who contributed to defining UML.

UML 1.1 Core Team

- *Hewlett-Packard Company*: Martin Griss
- *IBM Corporation*: Steve Cook, Jos Warmer
- *ICON Computing*: Desmond D'Souza
- *I-Logix*: Eran Gery, David Harel
- *MCI Systemhouse Corporation*: Cris Kobryn, Joaquin Miller
- *IntelliCorp and James Martin & Co.*: James Odell
- *ObjecTime Limited*: John Hogg, Bran Selic
- *Oracle Corporation*: Guus Ramackers
- *PLATINUM Technology Inc.*: Dilhar DeSilva
- *Rational Software*: Grady Booch, Ed Eykholt (project lead), Ivar Jacobson, Gunnar Overgaard, Karin Palmkvist, Jim Rumbaugh
- *Taskon A/S*: Trygve Reenskaug
- *Sterling Software*: John Cheesman, Keith Short
- *Unisys Corporation*: Sridhar Iyengar, GK Khalsa

UML 1.1 Semantics Task Force

During the final submission phase, a team was formed to focus on improving the formality of the UML 1.0 semantics, as well as incorporating additional ideas from the partners. Under the leadership of Cris Kobryn, this team was very instrumental in reconciling diverse viewpoints into a consistent set of semantics, as expressed in the revised *UML Semantics*. Other members of this team were Dilhar DeSilva, Martin Griss, Sridhar Iyengar, Eran Gery, Gunnar Overgaard, Karin Palmkvist, Guus Ramackers, Bran Selic, and Jos Warmer. Booch, Jacobson, and Rumbaugh provided their expertise to the team, as well.

Contributors and Supporters

We also acknowledge the contributions, influence, and support of the following individuals.

Jim Amsden, Hernan Astudillo, Colin Atkinson, Dave Bernstein, Philip A. Bernstein, Michael Blaha, Conrad Bock, Mike Bradley, Ray Buhr, Gary Cernosek, James Cerrato, Michael Jesse Chonoles, Magnus Christerson, Dai Clegg, Peter Coad, Derek Coleman, Ward Cunningham, Raj Datta, Mike Devlin, Philippe Desfray, Bruce Douglass, Staffan Ehnebom, Maria Ericsson, Johannes Ernst, Don Firesmith, Martin Fowler, Adam Frankl, Eric Gamma, Dipayan Gangopadhyay, Garth Gullekson, Rick Hargrove, Tim Harrison, Richard Helm, Brian Henderson-Sellers, Michael Hirsch, Bob Hodges, Glenn Hollowell, Yves Holvoet, Jon Hopkins, John Hsia, Ralph Johnson, Anneke Kleppe, Philippe Kruchten, Paul Kyzivat, Martin Lang, Grant Larsen, Reed Letsinger, Mary Loomis, Jeff MacKay, Robert Martin, Terrie McDaniel, Jim McGee, Bertrand Meyer, Mike Meier, Randy Messer, Greg Meyers, Fred Mol, Luis Montero, Paul Moskowitz, Andy Moss, Jan Pachl, Paul Patrick, Woody Pidcock, Bill Premerlani, Jeff Price, Jerri Pries, Terry Quatrani, Mats Rahm, George Reich, Rich Reitman, Rudolf M. Riess, Erick Rivas, Kenny Rubin, Jim Rye, Danny Sabbah, Tom Schultz, Ed Seidewitz, Gregson Siu, Jeff Sutherland, Dan Tasker, Dave Tropeano, Andy Trice, Dan Uhlar, John Vlissides, Larry Wall, Paul Ward, Alan Wills, Rebecca Wirfs-Brock, Bryan Wood, Ed Yourdon, and Steve Zeigler.

0.6 References

[Bock/Odell 94]	C. Bock and J. Odell, "A Foundation For Composition," Journal of Object-oriented Programming, October 1994.
[Booch et al.]	Grady Booch, Jim Rumbaugh, and Ivar Jacobson, Unified Modeling Language User Guide, ISBN: 0-201-57168-4, Addison Wesley, est. publication December 1997. See *www.awl.com/cp/uml/uml.html*.
[Cook 94]	S. Cook and J. Daniels, Designing Object Systems: Object-oriented Modelling with Syntropy, Prentice-Hall Object-Oriented Series, 1994.
[D’Souza 97a]	D. D’Souza and A. Wills, "Input for the OMG Submission," *www.iconcomp.com/catalysis*
[D’Souza 97b]	D. D’Souza and A. Wills, "Catalysis: Component and Framework based development" *www.iconcomp.com/catalysis*
[Fowler 97]	M. Fowler with K. Scott, UML Distilled: Applying the Standard Object Modeling Language, ISBN 0-201-32563-2, Addison-Wesely, 1997. *http://www.awl.com/cp/uml/uml.html*
[Griss 96]	M. Griss, Domain Engineering And Variability In The Reuse-Driven Software Engineering Business. Object Magazine. Dec 1996. (See *www.hpl.hp.com/reuse*)
[Harel 87]	D. Harel, "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming 8 (1987), 231-274.
[Harel 96a]	D. Harel and E. Gery, "Executable Object Modeling with Statecharts," Proc. 18th Int. Conf. Soft. Eng., Berlin, IEEE Press, March, 1996, pp. 246-257.
[Harel 96b]	D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," ACM Trans. Soft. Eng. Method 5:4 (Oct. 1996).

[Jacobson et al.]	Ivar Jacobson, Grady Booch, and Jim Rumbaugh, The Objectory Software Development Process, ISBN: 0-201-57169-2, Addison Wesley est. publication December 1997. See *www.awl.com/cp/uml/uml.html* and the "Rational Objectory Process" on *www.rational.com*.
[Malan 96]	R. Malan, D. Coleman, R. Letsinger et al, The Next Generation of Fusion, Fusion Newsletter, Oct 1996. (See *www.hpl.hp.com/fusion*.)
[Martin/Odell 95]	J. Martin and J. Odell, Object-oriented Methods, A Foundation, ISBN: 0-13-630856-2, Prentice Hall, 1995
[Ramackers 95]	Ramackers, G. and Clegg, D., "Object Business Modelling, requirements and approach" in Sutherland, J. and Patel, D. (eds.), Proceedings of the OOPSLA95 workshop on Business Object Design and Implementation, Springer Verlag, publication pending.
[Ramackers 96]	Ramackers, G. and Clegg, D., "Extended Use Cases and Business Objects for BPR," ObjectWorld UK '96, London, June 18-21, 1996.
[Rumbaugh et al.]	Jim Rumbaugh, Ivar Jacobson, and Grady Booch, Unified Modeling Language Reference Manual, ISBN: 0-201-30998-X, Addison Wesley, est. publication December 1997. See *www.awl.com/cp/uml/uml.html*.
[UML Web Sites]	*www.rational.com/uml* *uml.systemhouse.mci.com*

Table of Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

UML Summary 1

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

Contents

This chapter contains the following topics.

1.1 Overview

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

1.2 Primary Artifacts of the UML

What are the primary artifacts of the UML? This can be answered from two different perspectives: the UML definition itself and how it is used to produce project artifacts.

1.2.1 UML-defining Artifacts

To aid the understanding of the artifacts that constitute the Unified Modeling Language itself, this document consists of the UML Semantics, UML Notation Guide, and UML Extensions sections.

1.2.2 Development Project Artifacts

The choice of what models and diagrams one creates has a profound influence upon how a problem is attacked and how a corresponding solution is shaped. Abstraction, the focus on relevant details while ignoring others, is a key to learning and communicating. Because of this:

- Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient.
- Every model may be expressed at different levels of fidelity.
- The best models are connected to reality.

In terms of the views of a model, the UML defines the following graphical diagrams:

- use case diagram
- class diagram
- behavior diagrams:
  - statechart diagram
  - activity diagram
  - interaction diagrams:
    - sequence diagram
    - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views. These diagrams are further described in the UML Notation Guide (Section 3).

A frequently asked question has been, "Why doesn't UML support data-flow diagrams?" Simply put, data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm. Activity diagrams accomplish much of what people want from DFDs, and then some. Activity diagrams are also useful for modeling workflow.

1.3 Motivation to Define the UML

This section describes several factors motivating the UML and includes why modeling is essential, it highlights a few key trends in the software industry, and describes the issues caused by divergence of modeling approaches.

1.3.1 Why We Model

Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Good models are essential for communication among project teams and to assure architectural soundness. We build models of complex systems because we cannot comprehend any such system in its entirety. As the complexity of systems increase, so does the importance of good modeling techniques. There are many additional factors of a project's success, but having a rigorous modeling language standard is one essential factor. A modeling language must include:

- Model elements — fundamental modeling concepts and semantics
- Notation — visual rendering of model elements
- Guidelines — idioms of usage within the trade

In the face of increasingly complex systems, visualization and modeling become essential. The UML is a well-defined and widely accepted response to that need. It is the visual modeling language of choice for building object-oriented and component-based systems.

1.3.2 Industry Trends in Software

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software. We look for techniques to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns, and frameworks. We also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, we recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing, and fault tolerance. Development for the worldwide web makes some things simpler, but exacerbates these architectural problems.

Complexity will vary by application domain and process phase. One of the key motivations in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domains.

1.3.3 Prior to Industry Convergence

Prior to the UML, there was no clear leading modeling language. Users had to choose from among many similar modeling languages with minor difference in overall expressive power. Most of the modeling languages shared a set of commonly accepted concepts that are expressed slightly differently in various languages. This lack of agreement discouraged new users from entering the OO market and from doing OO modeling, without greatly expanding the power of modeling. Users longed for the industry to adopt one, or a very few, broadly supported modeling languages suitable for general-purpose usage.

Some vendors were discouraged from entering the OO modeling area because of the need to support many similar, but slightly different, modeling languages. In particular, the supply of add-on tools has been depressed because small vendors cannot afford to support many different formats from many different front-end modeling tools. It is important to the entire OO industry to encourage broadly based tools and vendors, as well as niche products that cater to the needs of specialized groups.

The perpetual cost of using and supporting many modeling languages motivated many companies producing or using OO technology to endorse and support the development of the UML.

While the UML does not guarantee project success, it does improve many things. For example, it significantly lowers the perpetual cost of training and retooling when changing between projects or organizations. It provides the opportunity for new integration between tools, processes, and domains. But most importantly, it enables developers to focus on delivering business value and gives them a paradigm to accomplish this.

1.4 Goals of the UML

The primary design goals of the UML are as follows:

- Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.
- Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- Integrate best practices.

These goals are discussed in detail below.

Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models

It is important that the OOAD standard supports a modeling language that can be used "out of the box" to do normal general-purpose modeling tasks. If the standard merely provides a meta-meta-description that requires tailoring to a particular set of modeling concepts, then it will not achieve the purpose of allowing users to exchange models without losing information or without imposing excessive work to map their models to a very abstract form. The UML consolidates a set of core modeling concepts that are generally accepted across many current methods and modeling tools. These concepts are needed in many or most large applications, although not every concept is needed in every part of every application. Specifying a meta-meta-level format for the concepts is not sufficient for model users, because the concepts must be made concrete for real modeling to occur. If the concepts in different application areas were substantially different, then such an approach might work, but the core concepts needed by most application areas are similar and should be supported directly by the standard without the need for another layer.

Provide extensibility and specialization mechanisms to extend the core concepts

OMG expects that the UML will be tailored as new needs are discovered and for specific domains. At the same time, we do not want to force the common core concepts to be redefined or re-implemented for each tailored area. Therefore, we believe that the extension mechanisms should support deviations from the common case, rather than being required to implement the core OOA&D concepts themselves. The core concepts should not be changed more than necessary. Users need to be able to

- build models using core concepts without using extension mechanisms for most normal applications,
- add new concepts and notations for issues not covered by the core,
- choose among variant interpretations of existing concepts, when there is no clear consensus, and
- specialize the concepts, notations, and constraints for particular application domains.

Be independent of particular programming languages and development processes

The UML must and can support all reasonable programming languages. It also must and can support various methods and processes of building models. The UML can support multiple programming languages and development methods without excessive difficulty.

Provide a formal basis for understanding the modeling language

Because users will use formality to help understand the language, it must be both precise and approachable; a lack of either dimension damages its usefulness. The formalisms must not require excessive levels of indirection or layering, use of low-level mathematical notations distant from the modeling domain, such as set-theoretic notation, or operational definitions that are equivalent to programming an implementation. The UML provides a formal definition of the static format of the model using a metamodel expressed in UML class diagrams. This is a popular and widely accepted formal approach for specifying the format of a model and directly leads to the implementation of interchange formats. UML expresses well-formedness constraints in precise natural language plus Object Constraint Language expressions. UML expresses the operational meaning of most constructs in precise natural language. The fully formal approach taken to specify languages such as Algol-68 was not approachable enough for most practical usage.

Encourage the growth of the OO tools market

By enabling vendors to support a standard modeling language used by most users and tools, the industry benefits. While vendors still can add value in their tool implementations, enabling interoperability is essential. Interoperability requires that models can be exchanged among users and tools without loss of information. This can only occur if the tools agree on the format and meaning of all of the relevant concepts. Using a higher meta-level is no solution unless the mapping to the user-level concepts is included in the standard.

Support higher-level development concepts such as collaborations, frameworks, patterns, and components

Clearly defined semantics of these concepts is essential to reap the full benefit of OO and reuse. Defining these within the holistic context of a modeling language is a unique contribution of the UML.

Integrate best practices

A key motivation behind the development of the UML has been to integrate the best practices in the industry, encompassing widely varying views based on levels of abstraction, domains, architectures, life cycle stages, implementation technologies, etc. The UML is indeed such an integration of best practices.

1.5 Scope of the UML

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.

First and foremost, the Unified Modeling Language fuses the concepts of Booch, OMT, and OOSE. The result is a single, common, and widely usable modeling language for users of these and other methods.

Second, the Unified Modeling Language pushes the envelope of what can be done with existing methods. As an example, the UML authors targeted the modeling of concurrent, distributed systems to assure the UML adequately addresses these domains.

Third, the Unified Modeling Language focuses on a standard modeling language, not a standard process. Although the UML must be applied in the context of a process, it is our experience that different organizations and problem domains require different processes. (For example, the development process for shrink-wrapped software is an interesting one, but building shrink-wrapped software is vastly different from building hard-real-time avionics systems upon which lives depend.) Therefore, the efforts concentrated first on a common metamodel (which unifies semantics) and second on a common notation (which provides a human rendering of these semantics). The UML authors promote a development process that is use-case driven, architecture centric, and iterative and incremental.

The UML specifies a modeling language that incorporates the object-oriented community's consensus on core modeling concepts. It allows deviations to be expressed in terms of its extension mechanisms. The Unified Modeling Language provides the following:

- Sufficient semantics and notation to address a wide variety of contemporary modeling issues in a direct and economical fashion.
- Sufficient semantics to address certain expected future modeling issues, specifically related to component technology, distributed computing, frameworks, and executability.
- Extensibility mechanisms so individual projects can extend the metamodel for their application at low cost. We don't want users to adjust the UML metamodel itself.
- Extensibility mechanisms so that future modeling approaches could be grown on top of the UML.
- Sufficient semantics to facilitate model interchange among a variety of tools.
- Sufficient semantics to specify the interface to repositories for the sharing and storage of model artifacts.

1.5.1 Outside the Scope of the UML

Programming Languages

The UML, a visual modeling language, is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, but it does draw the line as you move toward code. For example, complex branches and joins are better expressed in a textual programming language. The UML does have a tight mapping to a family of OO languages so that you can get the best of both worlds.

Tools

Standardizing a language is necessarily the foundation for tools and process. Tools and their interoperability are very dependent on a solid semantic and notation definition, such as the UML provides. The UML defines a semantic metamodel, not a tool interface, storage, or run-time model, although these should be fairly close to one another.

The UML documents do include some tips to tool vendors on implementation choices, but do not address everything needed. For example, they don't address topics like diagram coloring, user navigation, animation, storage/implementation models, or other features.

Process

Many organizations will use the UML as a common language for its project artifacts, but will use the same UML diagram types in the context of different processes. The UML is intentionally process independent, and defining a standard process was not a goal of the UML or OMG's RFP.

The UML authors do recognize the importance of process. The presence of a well-defined and well-managed process is often a key discriminator between hyperproductive projects and unsuccessful ones. The reliance upon heroic programming is not a sustainable business practice. A process

- provides guidance as to the order of a team's activities,
- specifies what artifacts should be developed,
- directs the tasks of individual developers and the team as a whole, and
- offers criteria for monitoring and measuring a project's products and activities.

Processes by their very nature must be tailored to the organization, culture, and problem domain at hand. What works in one context (shrink-wrapped software development, for example) would be a disaster in another (hard-real-time, human-rated systems, for example). The selection of a particular process will vary greatly, depending on such things as problem domain, implementation technology, and skills of the team.

Booch, OMT, OOSE, and many other methods have well-defined processes, and the UML can support most methods. There has been some convergence on development process practices, but there is not yet consensus for standardization. What will likely result is general agreement on best practices and potentially the embracing of a process framework, within which individual processes can be instantiated. Although the UML does not mandate a process, its developers have recognized the value of a use-case driven, architecture-centric, iterative, and incremental process, so were careful to enable (but not require) this with the UML.

1.5.2 Comparing UML to Other Modeling Languages

It should be made clear that the Unified Modeling Language is not a radical departure from Booch, OMT, or OOSE, but rather the legitimate successor to all three. This means that if you are a Booch, OMT, or OOSE user today, your training, experience, and tools will be preserved, because the Unified Modeling Language is a natural evolutionary step. The UML will be equally easy to adopt for users of many other methods, but their authors must decide for themselves whether to embrace the UML concepts and notation underneath their methods.

The Unified Modeling Language is more expressive yet cleaner and more uniform than Booch, OMT, OOSE, and other methods. This means that there is value in moving to the Unified Modeling Language, because it will allow projects to model things they could not have done before. Users of most other methods and modeling languages will gain value by moving to the UML, since it removes the unnecessary differences in notation and terminology that obscure the underlying similarities of most of these approaches.

With respect to other visual modeling languages, including entity-relationship modeling, BPR flow charts, and state-driven languages, the UML should provide improved expressiveness and holistic integrity.

Users of existing methods will experience slight changes in notation, but this should not take much relearning and will bring a clarification of the underlying semantics. If the unification goals have been achieved, UML will be an obvious choice when beginning new projects, especially as the availability of tools, books, and training becomes widespread. Many visual modeling tools support existing notations, such as Booch, OMT, OOSE, or others, as views of an underlying model; when these tools add support for UML (as some already have) users will enjoy the benefit of switching their current models to the UML notation without loss of information.

Existing users of any OO method can expect a fairly quick learning curve to achieve the same expressiveness as they previously knew. One can quickly learn and use the basics productively. More advanced techniques, such as the use of stereotypes and properties, will require some study since they enable very expressive and precise models needed only when the problem at hand requires them.

1.5.3 Features of the UML

The goals of the unification efforts were to keep it simple, to cast away elements of existing Booch, OMT, and OOSE that didn’t work in practice, to add elements from other methods that were more effective, and to invent new only when an existing solution was not available. Because the UML authors were in effect designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and over-engineering (having an icon for every conceivable modeling element). To that end, they were very careful about adding new things, because they didn’t want to make the UML unnecessarily complex. Along the way, however, some things were found that were advantageous to add because they have proven useful in practice in other modeling.

There are several new concepts that are included in UML, including

- extensibility mechanisms (stereotypes, tagged values, and constraints),
- threads and processes,
- distribution and concurrency (e.g., for modeling ActiveX/DCOM and CORBA),
- patterns/collaborations,
- activity diagrams (for business process modeling),
- refinement (to handle relationships between levels of abstraction),

- interfaces and components, and
- a constraint language.

Many of these ideas were present in various individual methods and theories but UML brings them together into a coherent whole. In addition to these major changes, there are many other localized improvements over the Booch, OMT, and OOSE semantics and notation.

The UML is an evolution from Booch, OMT, OOSE, other object-oriented methods, and many other sources. These various sources incorporated many different elements from many authors, including non-OO influences. The UML notation is a melding of graphical syntax from various sources, with a number of symbols removed (because they were confusing, superfluous, or little used) and with a few new symbols added. The ideas in the UML come from the community of ideas developed by many different people in the object-oriented field. The UML developers did not invent most of these ideas; rather, their role was to select and integrate the best ideas from OO and computer-science practices. The actual genealogy of the notation and underlying detailed semantics is complicated, so it is discussed here only to provide context, not to represent precise history.

Use-case diagrams are similar in appearance to those in OOSE.

Class diagrams are a melding of OMT, Booch, class diagrams of most other OO methods. Extensions (e.g., stereotypes and their corresponding icons) can be defined for various diagrams to support other modeling styles. Stereotypes, constraints, and taggedValues are concepts added in UML that did not previously exist in the major modeling languages.

Statechart diagrams are substantially based on the statecharts of David Harel with minor modifications. The Activity diagram, which shares much of the same underlying semantics, is similar to the work flow diagrams developed by many sources including many pre-OO sources.

Sequence diagrams were found in a variety of OO methods under a variety of names (interaction, message trace, and event trace) and date to pre-OO days. Collaboration diagrams were adapted from Booch (object diagram), Fusion (object interaction graph), and a number of other sources.

Collaborations are now first-class modeling entities, and often form the basis of patterns.

The implementation diagrams (component and deployment diagrams) are derived from Booch's module and process diagrams, but they are now component-centered, rather than module-centered and are far better interconnected.

Stereotypes are one of the extension mechanisms and extend the semantics of the metamodel. User-defined icons can be associated with given stereotypes for tailoring the UML to specific processes.

Object Constraint Language is used by UML to specify the semantics and is provided as a language for expressions during modeling. OCL is an expression language having its root in the Syntropy method and has been influenced by expression languages in other methods like Catalysis. The informal navigation from OMT has the same intent, where OCL is formalized and more extensive.

Each of these concepts has further predecessors and many other influences. We realize that any brief list of influences is incomplete and we recognize that the UML is the product of a long history of ideas in the computer science and software engineering area.

1.6 UML - Past, Present, and Future

The UML was developed by Rational Software and its partners. Many companies are incorporating the UML as a standard into their development process and products, which cover disciplines such as business modeling, requirements management, analysis & design, programming, and testing.

1.6.1 UML 0.8 - 0.91

Precursors to UML

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. Several other techniques influenced these languages, including Entity-Relationship modeling, the Specification & Description Language (SDL, circa 1976, CCITT), and other techniques. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear, most notably Booch ’93, the continued evolution of OMT, and Fusion. These methods began to incorporate each other’s techniques, and a few clearly prominent methods emerged, including the OOSE, OMT-2, and Booch ’93 methods. Each of these was a complete method, and was recognized as having certain strengths. In simple terms, OOSE was a use-case oriented approach that provided excellent support business engineering and requirements analysis. OMT-2 was especially expressive for analysis and data-intensive information systems. Booch ’93 was particularly expressive during design and construction phases of projects and popular for engineering-intensive applications.

Booch, Rumbaugh, and Jacobson Join Forces

The development of UML began in October of 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. Given that the Booch and OMT methods were already independently growing together and were collectively recognized as leading object-oriented methods worldwide, Booch and Rumbaugh joined forces to forge a complete unification of their work. A draft version 0.8 of the

Unified Method, as it was then called, was released in October of 1995. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. The Objectory name is now used within Rational primarily to describe its UML-compliant process, the Rational Objectory Process.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

As they began their unification, they established four goals to focus their efforts:

1. Enable the modeling of systems (and not just software) using object-oriented concepts
2. Establish an explicit coupling to conceptual as well as executable artifacts
3. Address the issues of scale inherent in complex, mission-critical systems
4. Create a modeling language usable by both humans and machines

Devising a notation for use in object-oriented analysis and design is not unlike designing a programming language. There are tradeoffs. First, one must bound the problem: Should the notation encompass requirement specification? (Yes, partially.) Should the notation extend to the level of a visual programming language? (No.) Second, one must strike a balance between expressiveness and simplicity: Too simple a notation will limit the breadth of problems that can be solved; too complex a notation will overwhelm the mortal developer. In the case of unifying existing methods, one must also be sensitive to the installed base: Make too many changes, and you will confuse existing users. Resist advancing the notation, and you will miss the opportunity of engaging a much broader set of users. The UML definition strives to make the best tradeoffs in each of these areas.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

1.6.2 UML Partners

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML definition. Those contributing most to the UML definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML, a modeling language that was well defined, expressive, powerful, and generally applicable.

In January 1997 IBM & ObjecTime; Platinum Technology; Ptech; Taskon & Reich Technologies; and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners.

This document is based on the UML 1.1 release and is the result of a collaborative team effort. The UML Partners have worked hard as a team to define UML. While each partner came in with their own perspective and areas of interest, the result has benefited from each of them and from the diversity of their experiences. The UML Partners contributed a variety of expert perspectives, including, but not limited to, the following: OMG and RM-ODP technology perspectives, business modeling, constraint language, state machine semantics, types, interfaces, components, collaborations, refinement, frameworks, distribution, and metamodel.

1.6.3 UML - Present and Future

The UML is nonproprietary and open to all. It addresses the needs of user and scientific communities, as established by experience with the underlying methods on which it is based. Many methodologists, organizations, and tool vendors have committed to use it. Since the UML builds upon similar semantics and notation from Booch, OMT, OOSE, and other leading methods and has incorporated input from the UML partners and feedback from the general public, widespread adoption of the UML should be straightforward.

There are two aspects of "unified" that the UML achieves: First, it effectively ends many of the differences, often inconsequential, between the modeling languages of previous methods. Secondly, and perhaps more importantly, it unifies the perspectives among many different kinds of systems (business versus software), development phases (requirements analysis, design, and implementation), and internal concepts.

Standardization of the UML

Many organizations have already endorsed the UML as their organization’s standard, since it is based on the modeling languages of leading OO methods. The UML is ready for widespread use. This document is suitable as the primary source for authors writing books and training materials, as well as developers implementing visual modeling tools. Additional collateral, such as articles, training courses, examples, and books, will soon make the UML very approachable for a wide audience.

Industrialization

Many organizations and vendors worldwide have already embraced the UML. The number of endorsing organizations is expected to grow significantly over time. These organizations will continue to encourage the use of the Unified Modeling Language by making the definition readily available and by encouraging other methodologists, tool vendors, training organizations, and authors to adopt the UML.

The real measure of the UML's success is its use on successful projects and the increasing demand for supporting tools, books, training, and mentoring.

Future UML Evolution

Although the UML defines a precise language, it is not a barrier to future improvements in modeling concepts. We have addressed many leading-edge techniques, but expect additional techniques to influence future versions of the UML. Many advanced techniques can be defined using UML as a base. The UML can be extended without redefining the UML core.

The UML, in its current form, is expected to be the basis for many tools, including those for visual modeling, simulation, and development environments. As interesting tool integrations are developed, implementation standards based on the UML will become increasingly available.

The UML has integrated many disparate ideas, so this integration will accelerate the use of OO. Component-based development is an approach worth mentioning. It is synergistic with traditional object-oriented techniques. While reuse based on components is becoming increasingly widespread, this does not mean that component-based techniques will replace object-oriented techniques. There are only subtle differences between the semantics of components and classes.

UML Semantics 2

The UML Semantics section is primarily intended as a comprehensive and precise specification of the UML's semantic constructs.

Contents

This chapter contains the following sections.

Part 1 - Background

2.1 Introduction

2.1.1 Purpose and Scope

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.

This section 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.

This section provides complete semantics for all modeling notations described in the UML Notation Guide (Chapter 3). 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 Guide includes a summary of the semantics sections that are relevant to each diagram technique.

2.1.2 Approach

This section 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 “Language Architecture” on page 2-4.

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 “Language Architecture” on page 2-4.

The metamodel is described in a semi-formal manner using these 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 “Language Formalism” on page 2-7.

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.

2.2 Language Architecture

2.2.1 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 2-1 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 layer are: <Acme_Software_Share_98789>, 654.56, sell_limit_order, and <Stock_Quote_Svr_32123>.

The UML metamodel has been architected so that it 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 “Architectural Alignment with Other Technologies” in the Preface.

2.2.2 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 2-1 on page 2-6.

---

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

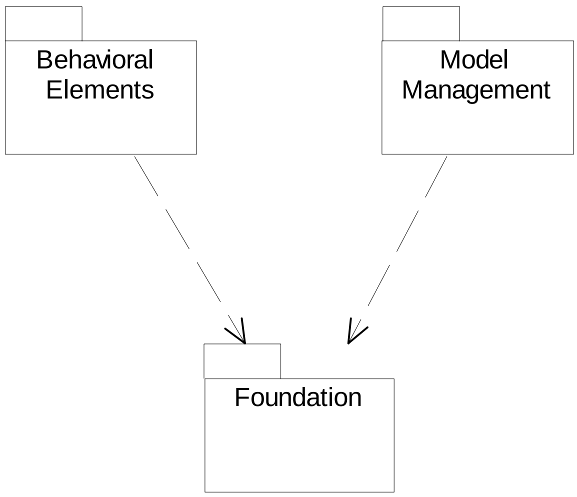


Figure 2-1 Top-Level Packages

The Foundation and Behavioral Elements packages are further decomposed as shown in Figure 2-2 and Figure 2-3 on page 2-7.

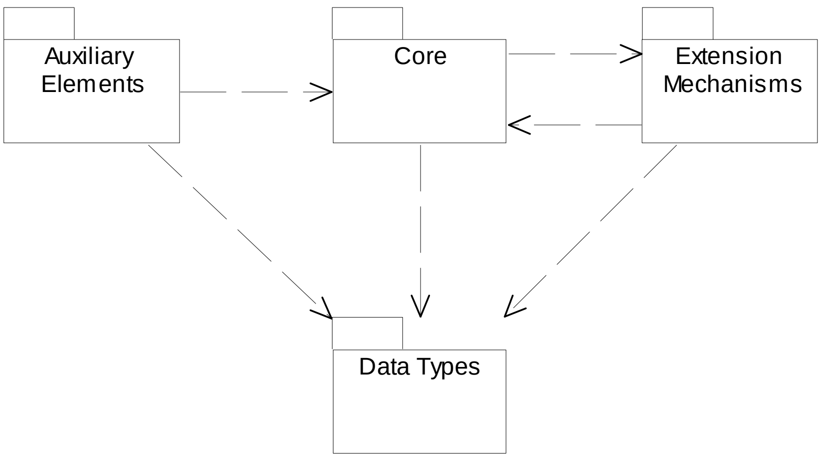


Figure 2-2 Foundation Packages

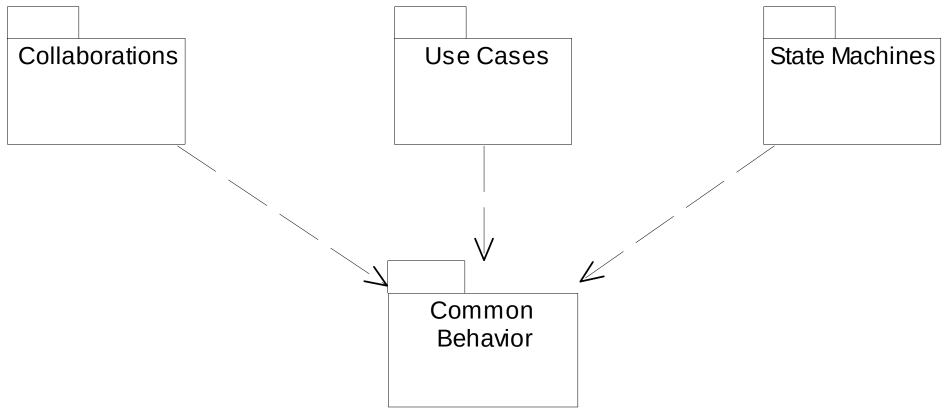


Figure 2-3 Behavioral Elements Packages

The functions and contents of these packages are described in this chapter’s Part 3, Behavioral Elements.

2.3 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.

2.3.1 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 the Abstract Syntax sections.

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.

2.3.2 Package Specification Structure

This section provides information for each package in the UML metamodel. Each package has one or more of the following subsections.

---

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.

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.

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.

2.3.3 Use of a Constraint Language

The specification uses the Object Constraint Language (OCL), as defined in Object Constraint Language Specification (Chapter 4), 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.

2.3.4 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).

2.3.5 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.
- Metaclass 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').
- 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 stereotypes are delimited by guillemets and begin with lowercase (e.g., «type»).

Part 2 - Foundation Packages

The Foundation package is the infrastructure for UML. The Foundation package is decomposed into several subpackages: Core, Auxiliary Elements, Extension Mechanisms, and Data Types.

2.4 Overview

Figure 2-4 illustrates the Foundation Packages. 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.

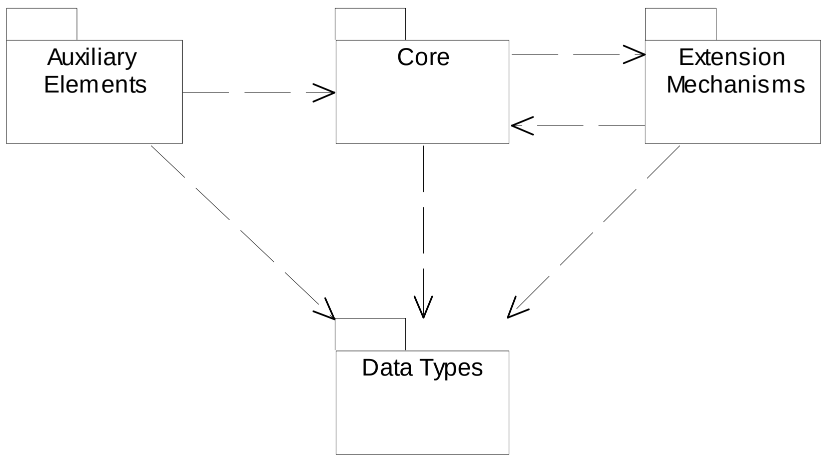


Figure 2-4 Foundation Packages

2.5 Core

2.5.1 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.

2.5.2 Abstract Syntax

The abstract syntax for the Core package is expressed in graphic notation in the following figures. Figure 2-5 on page 2-13 shows the model elements that form the structural backbone of the metamodel. Figure 2-6 on page 2-14 shows the model elements that define relationships.

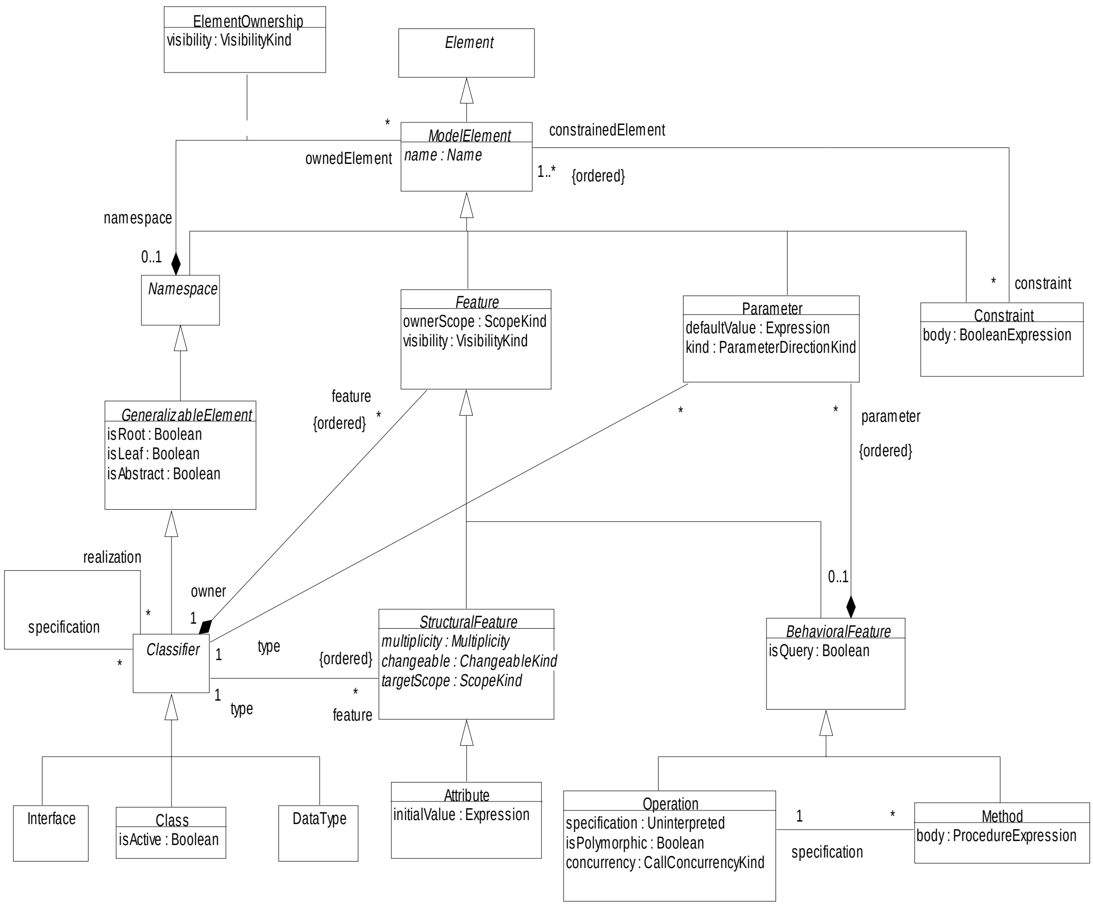


Figure 2-5 Core Package - Backbone

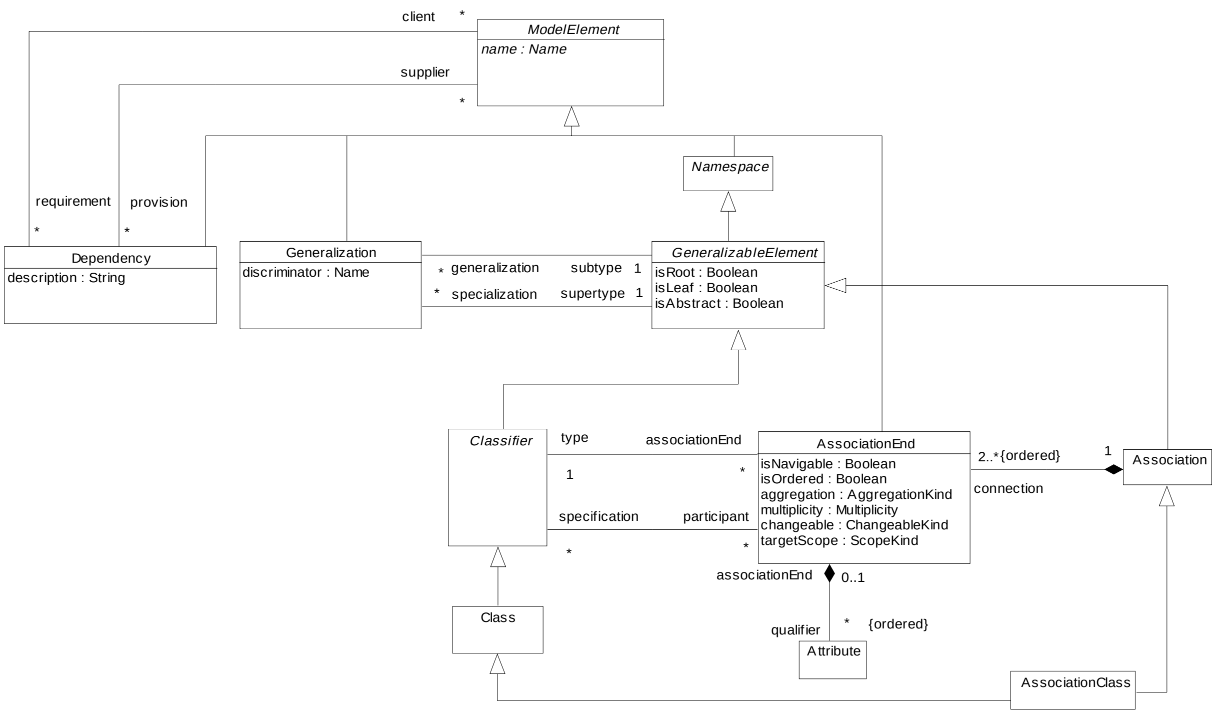


Figure 2-6 Core Package - Relationships

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; therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.
- composite - The end is a composite; therefore, the other end is 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.

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 line 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.
initial value	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 supersede 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. 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" on page 2-37 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 requires 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” on page 2-25.

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 a 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 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 ProcedureExpression.

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

owned	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 include:

- 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 correctly 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 Uninterpreted.

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. For example, it specifies 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.

2.5.3 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 three 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 Classifie 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 GeneralizableElement (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 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)
```

2.5.4 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” on page 2-38). 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 supersedes 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” on page 2-38.

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 be 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

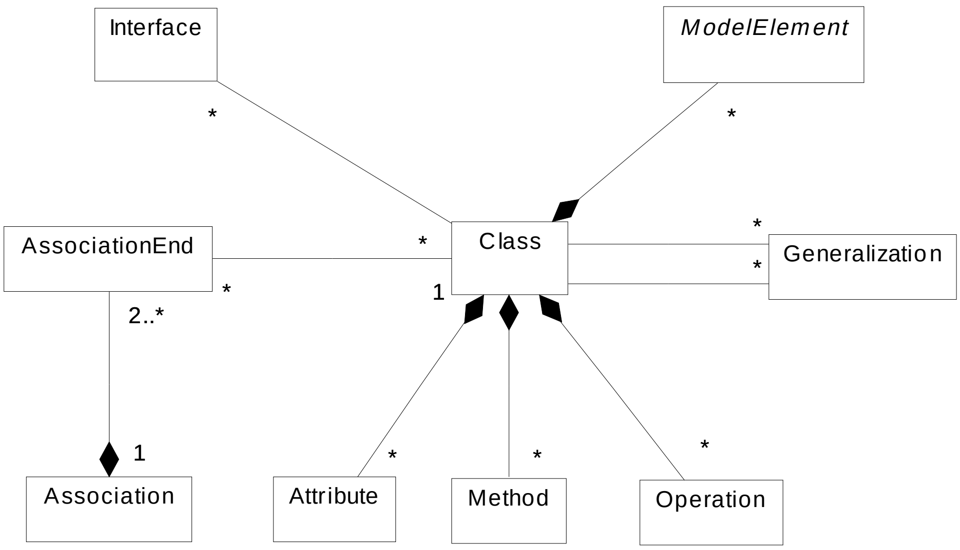


Figure 2-7 Class Illustration

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.

Note – 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 to:

- none - no constraints exists,
- frozen - the value cannot be replaced or added to once it has been initialized, or
- addOnly - new values may be added to a set but not removed or altered.

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 if 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* on the next page. 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; however, if the visibility is private, then the element is not visible and therefore not available in the subclass.

Interface

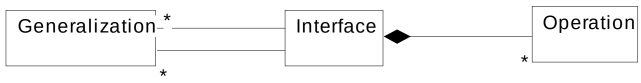


Figure 2-8 Interface Illustration

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 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

Figure 2-9 Association Illustration

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). For instance, 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 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 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

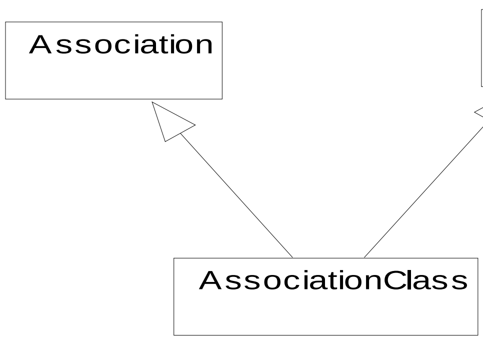


Figure 2-10 AssociationClass Illustration

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

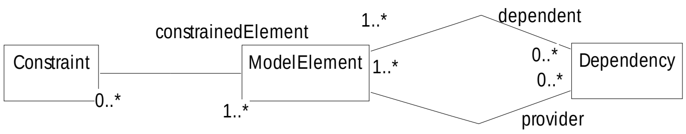


Figure 2-11 Miscellaneous Illustration

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 probably must 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, similar to class, is data type; however, 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. Usually, it is used for specification of the type of an attribute. An enumeration type is a user-definable type comprising a finite number of values.

2.5.5 Standard Elements

The predefined stereotypes, constraints, and tagged values for the Core package are listed in Table 2-2 and defined in Appendix A - UML Standard Elements.

Table 2-2 Core - Standard Elements

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

2.5.6 Notes

In UML, Associations can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) 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. The Method constructs are defined in the corresponding subclasses of BehavioralFeature.

A Usage or Binding dependency can be established only 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.

Note – The 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.

2.6 Auxiliary Elements

2.6.1 Overview

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

2.6.2 Abstract Syntax

The abstract syntax for the Auxiliary Elements package is expressed in graphic notation in the following figures.

Auxiliary Elements: Dependencies

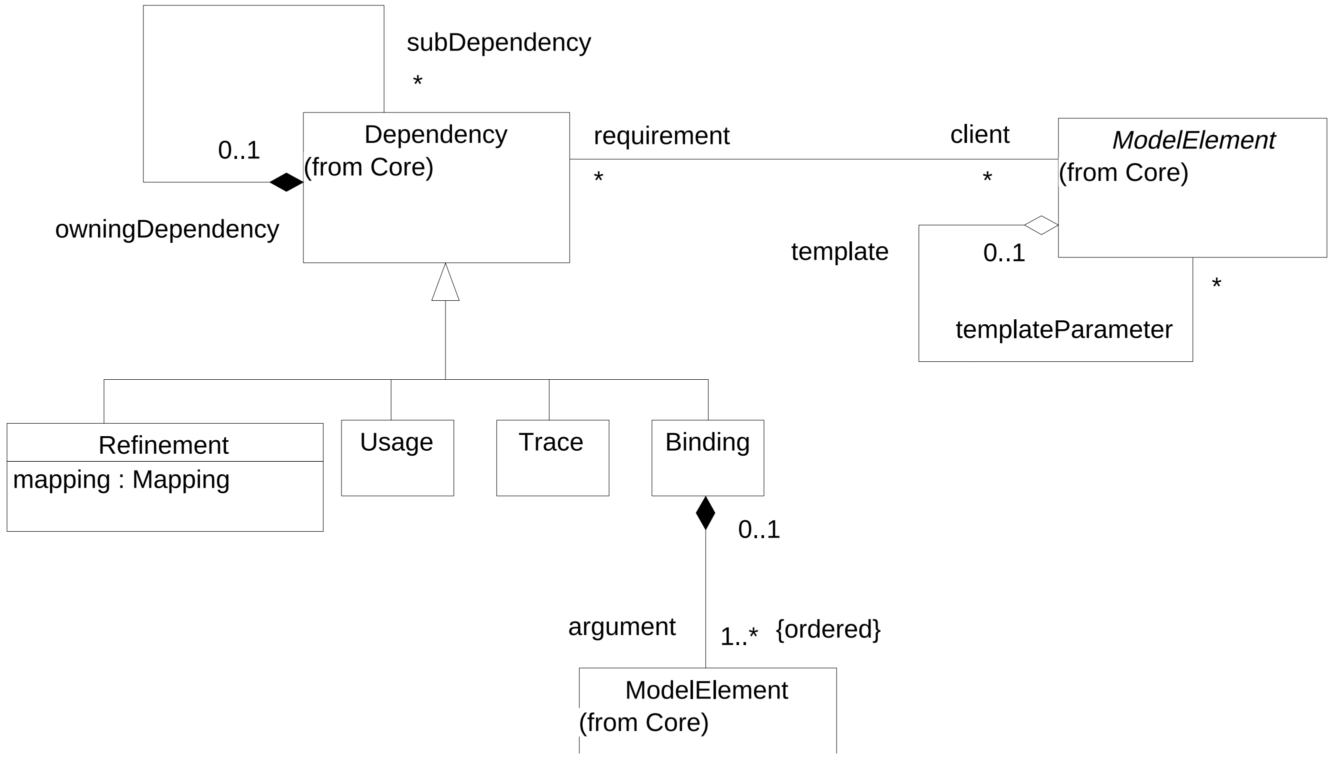


Figure 2-12 Auxiliary Elements - Dependencies and Templates

Auxiliary Elements:
Physical Structures and View Elements

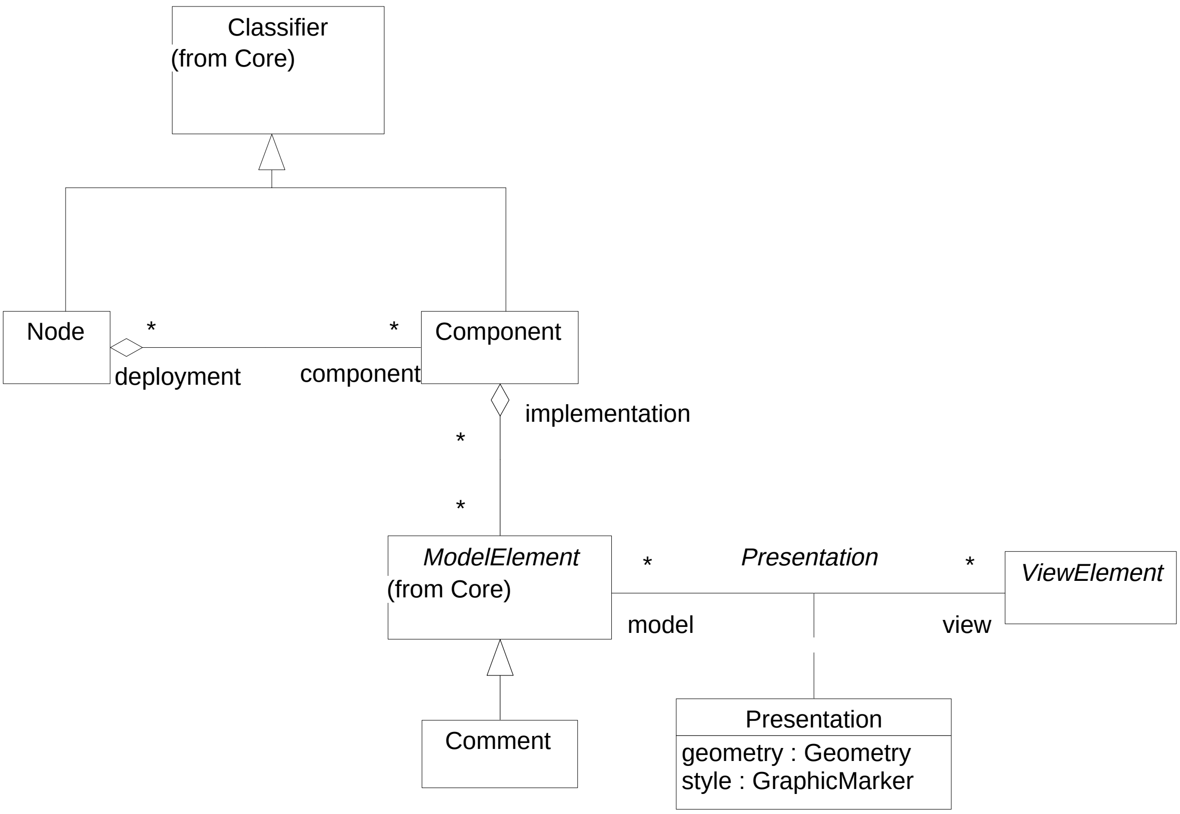


Figure 2-13 Auxiliary Elements - Physical Structures and View Elements

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 Classifier. 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	The 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 still has 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 Classifier. 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.

Attributes

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 denote dependencies at specification level, rather than runtime dependencies; therefore, 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.

2.6.3 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 it has parameters.

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

[2] A ModelElement is an instantiated template when it is 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.

2.6.4 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 possible 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.

2.6.5 Standard Elements

The predefined stereotypes, constraints and tagged values for the Auxiliary Elements package are listed in Table 2-3 and defined in Appendix A - UML Standard Elements.

Table 2-3 Auxiliary Elements - Standard Elements

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

2.7 Extension Mechanisms

2.7.1 Overview

The Extension Mechanisms package is the subpackage 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:

- To add new modeling elements for use in creating UML models.

- To define standard items that are not considered interesting or complex enough to be defined directly as UML metamodel elements.
- To define process-specific or implementation language-specific extensions.
- 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 (MOF).

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 augment the classification mechanism based on the built in UML metamodel class hierarchy; 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 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.

2.7.2 Abstract Syntax

The abstract syntax for the Extension Mechanisms package is expressed in graphic notation in Figure 2-14 on page 2-58.

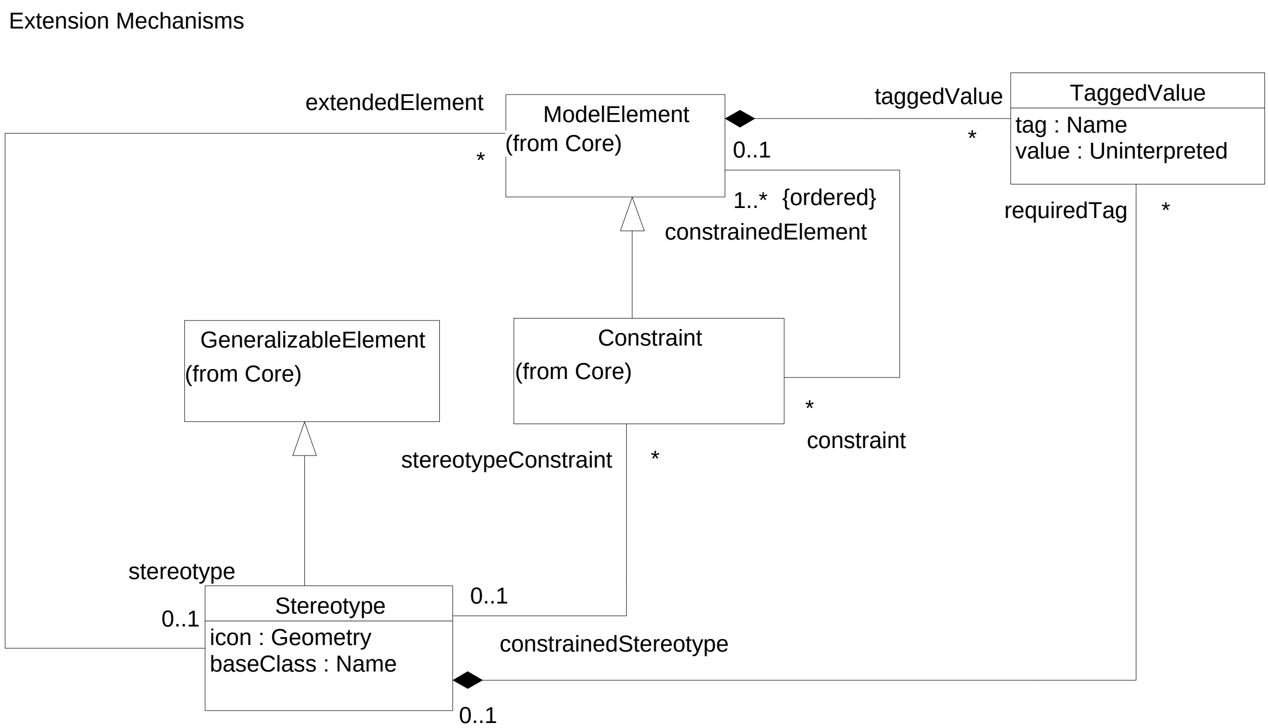


Figure 2-14 Extension Mechanisms

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 these 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. 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	ATaggedValue that is attached to a Stereotype. A particular TaggedValue can be attached to either a ModelElement or a Stereotype, but not both.

2.7.3 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 baseClass) 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 and 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 metaattributes 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.required Tag.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.

2.7.4 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.

2.7.5 Standard Elements

None.

2.7.6 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 Section 5.2.2, “Mapping of Interface Model into MOF” for a discussion of the relationship of the UML to the OMG Meta Object Facility (MOF).

2.8 Data Types

2.8.1 Overview

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

2.8.2 Abstract Syntax

The abstract syntax for the Data Types package is expressed in graphic notation in Figure 2-15 on page 2-66.

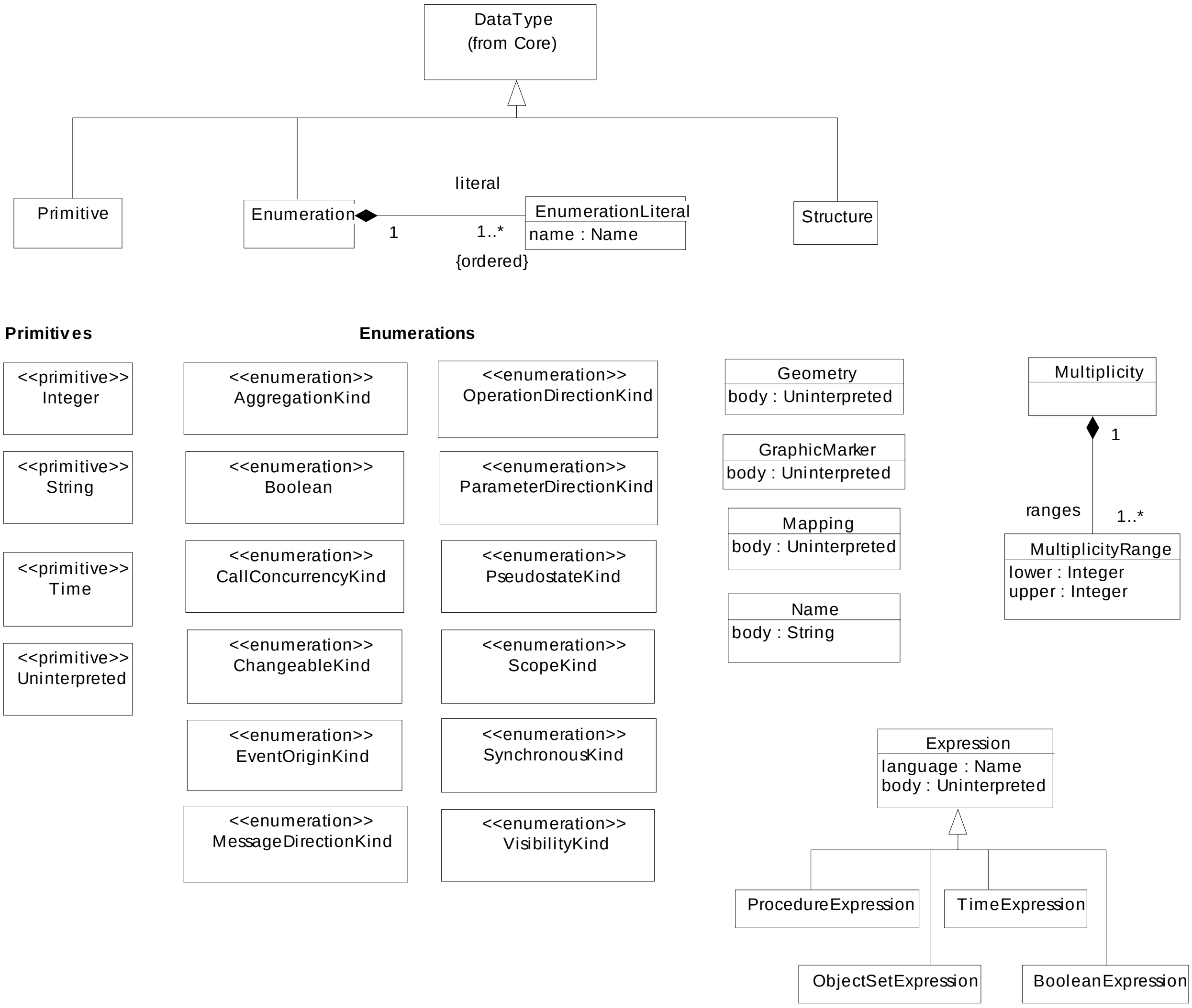


Figure 2-15 Data Types

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 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 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, PseudostateKind 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 String 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.

2.8.3 Standard Elements

The predefined stereotypes, constraints and tagged values for the Data Types package are listed in Table 2-4 and defined in Appendix A - UML Standard Elements.

Table 2-4 Data Types - Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
DataType	«enumeration»		

Part 3 - Behavioral Elements

This section 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.

2.9 Overview

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.

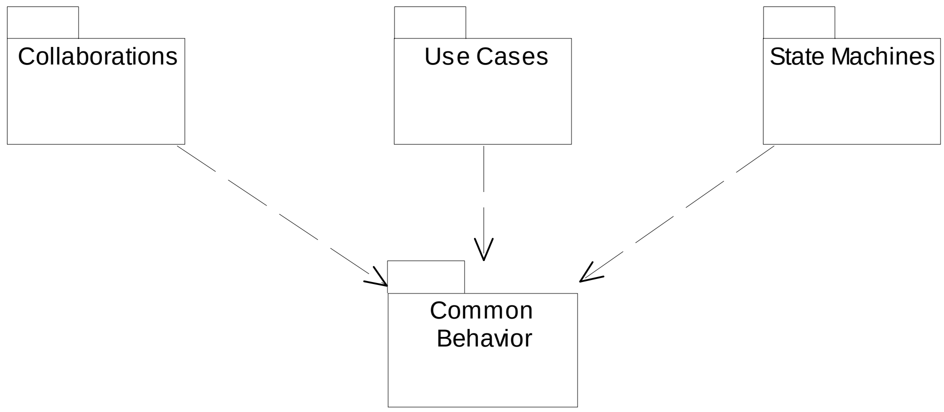


Figure 2-16 Behavioral Elements Package

2.10 Common Behavior

2.10.1 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.

2.10.2 Abstract Syntax

The abstract syntax for the Common Behavior package is expressed in graphic notation in the following figures. Figure 2-17 on page 2-72 shows the model elements that define Requests, which include Signals and Operations.

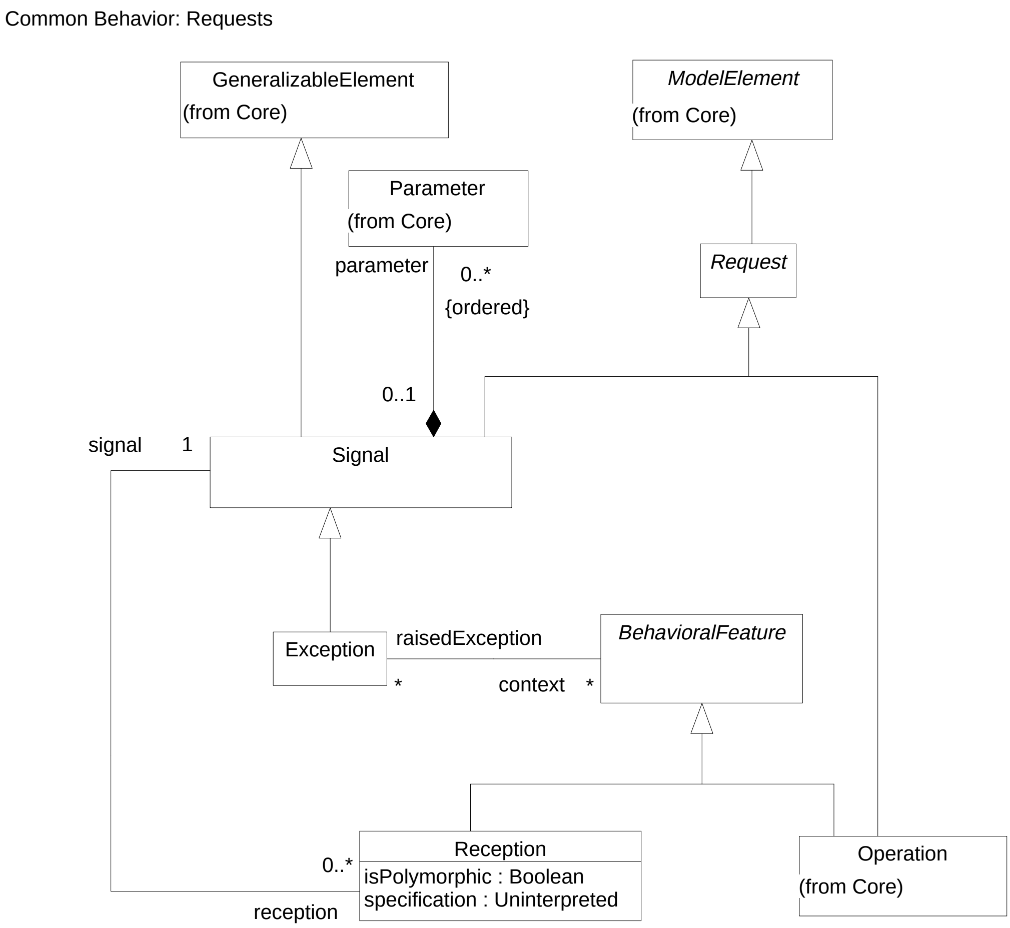


Figure 2-17 Common Behavior Requests

Figure 2-18 on page 2-73 illustrates the model elements that specify various actions, such as CreateAction, CallAction and SendAction.

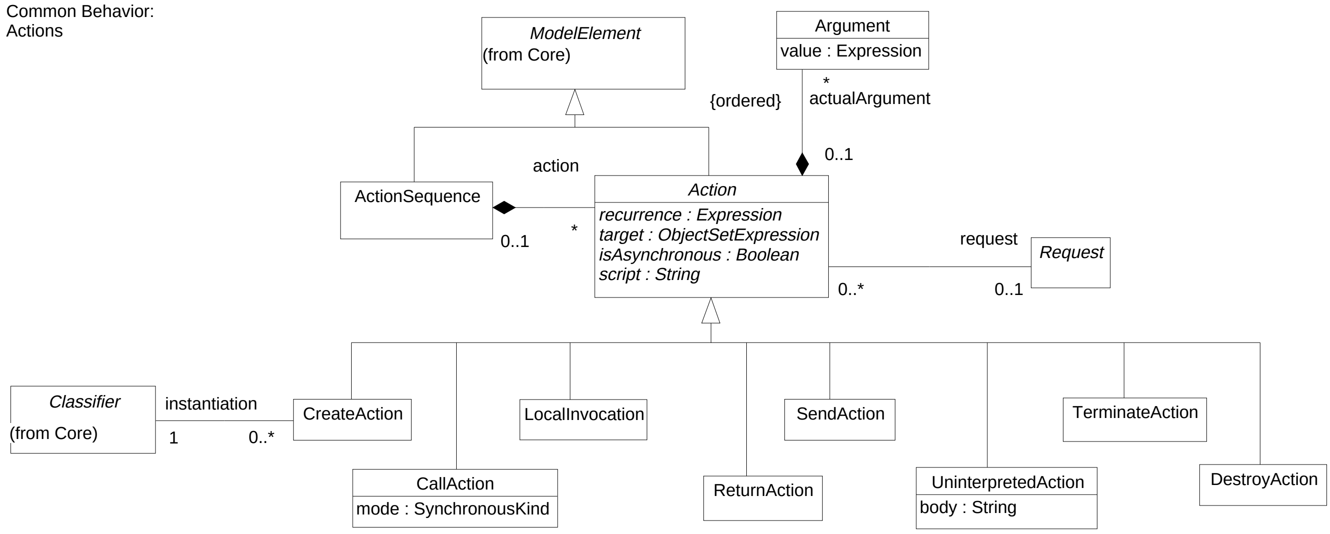


Figure 2-18 Common Behavior - Actions

Figure 2-19 on page 2-74 shows the model elements that define Instances and Links.

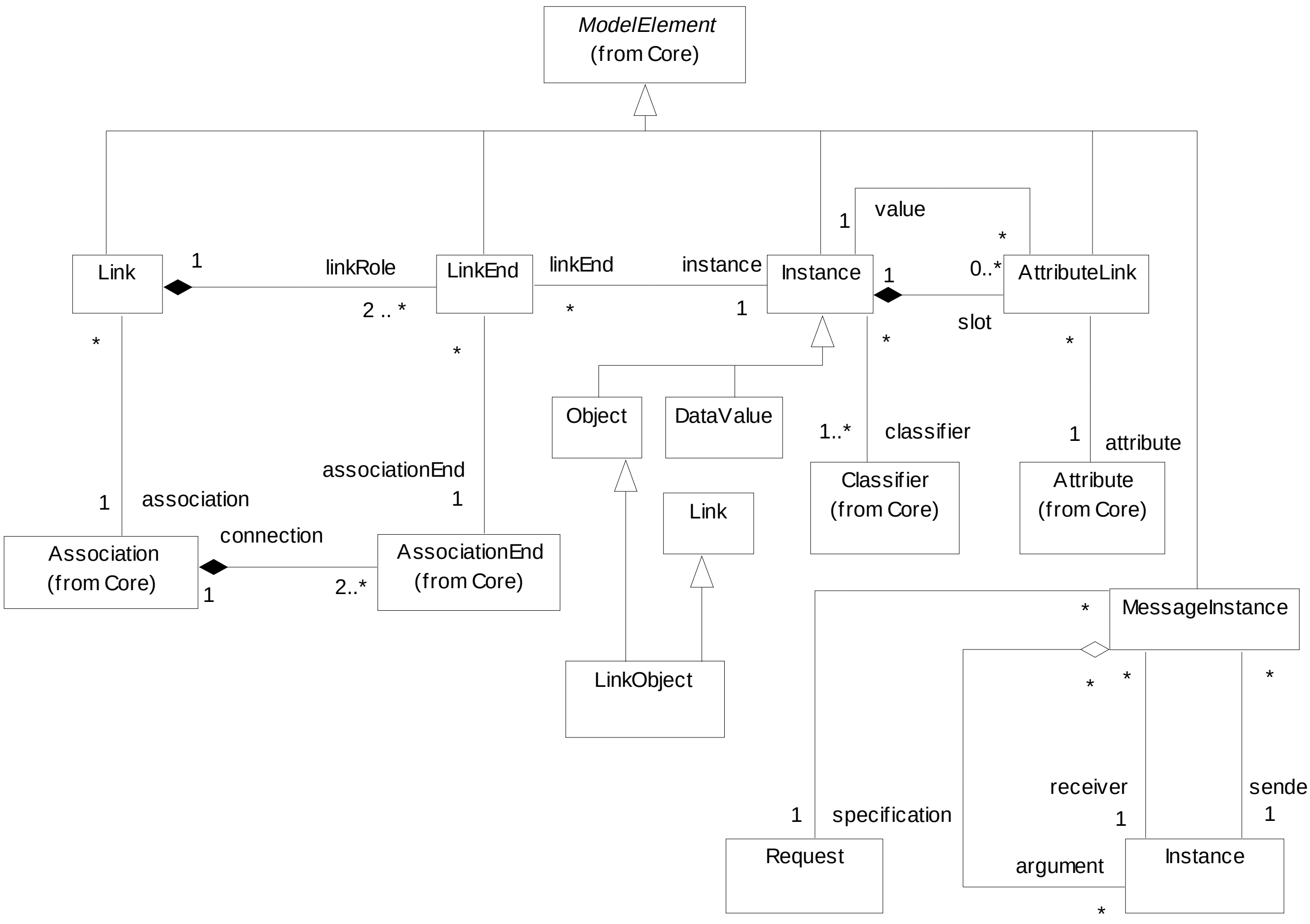


Figure 2-19 Common Behavior - Instances and Links

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, or 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 that 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 generalizable 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.

2.10.3 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 refers 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
```

2.10.4 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 fulfills 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.

2.10.5 Standard Elements

The predefined stereotypes, constraints and tagged values for the Common Behavior package are listed in Table 2-5 and defined in Appendix A - UML Standard Elements.

Table 2-5 Common Behavior - Standard Elements

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

2.11 Collaborations

2.11.1 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.

2.11.2 Abstract Syntax

The abstract syntax for the Collaborations package is expressed in graphic notation in Figure 2-20.

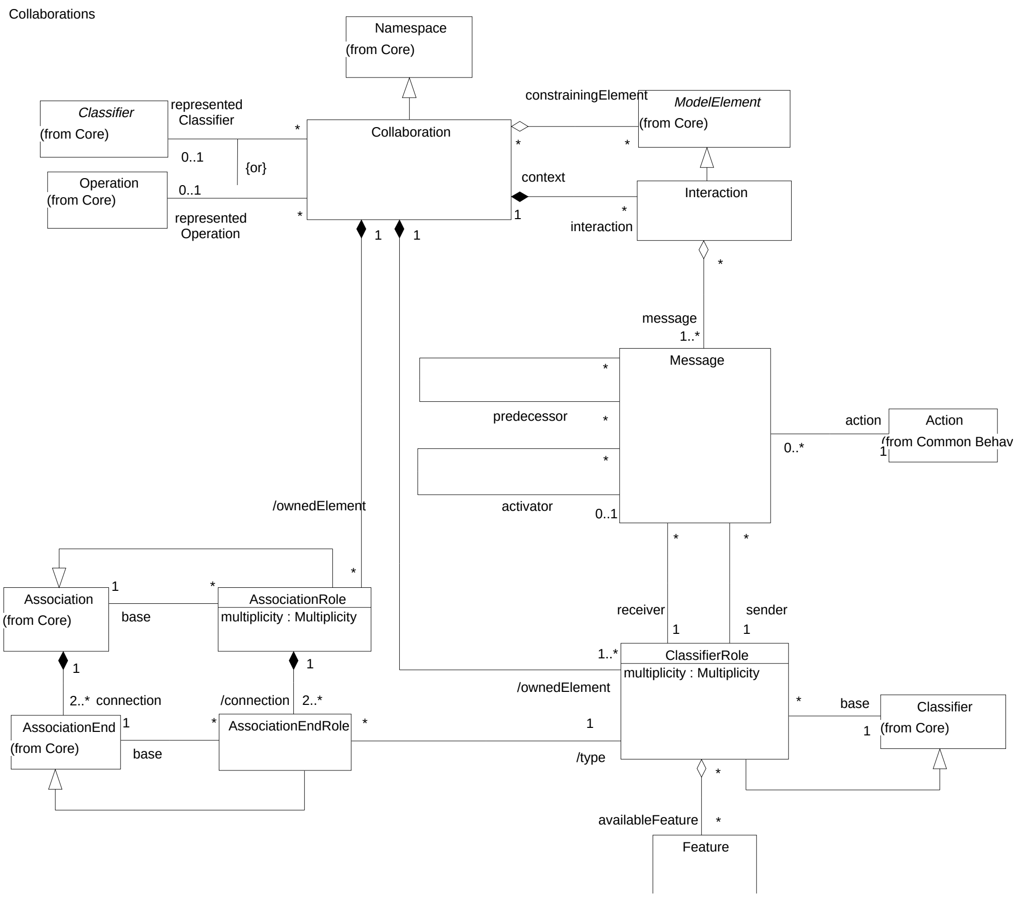


Figure 2-20 Collaborations

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 that 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

action	The specification of the Message.
activator	The Message that called the operation whose method contains the current 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.

2.11.3 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)
```

2.11.4 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 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).

2.11.5 Standard Elements

None.

2.11.6 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.

2.12 Use Cases

2.12.1 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.

2.12.2 Abstract Syntax

The abstract syntax for the Use Cases package is expressed in graphic notation in Figure 2-21 on page 2-99.

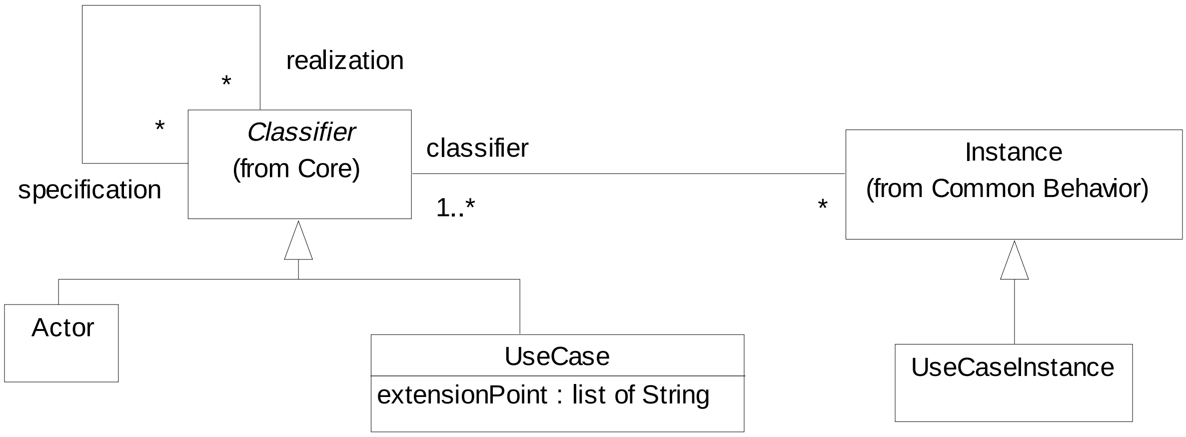


Figure 2-21 Use Cases

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.

2.12.3 Well-FormednessRules

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)
) )
```

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.

2.12.4 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

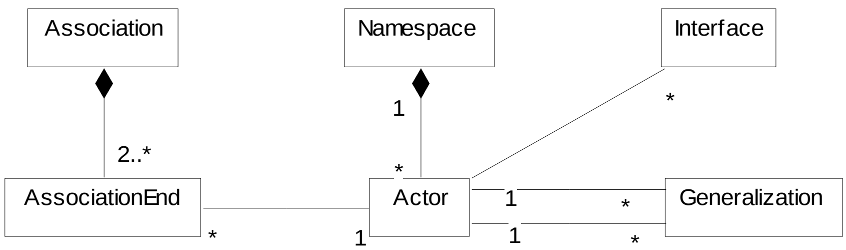


Figure 2-22 Actor Illustration

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

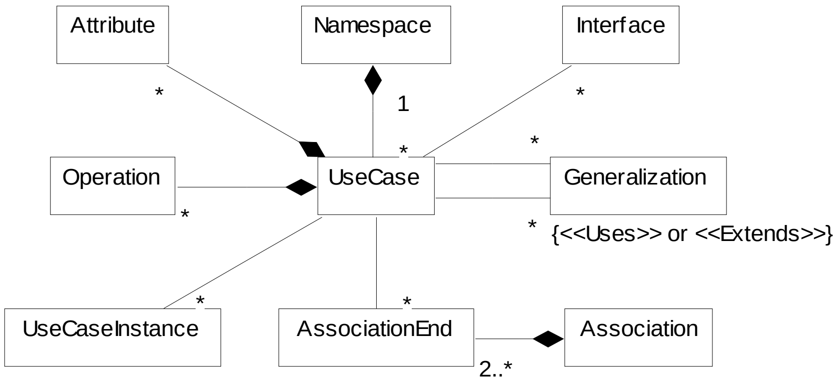


Figure 2-23 UseCase Illustration

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. 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.

2.12.5 Standard Elements

See Appendix A - UML Standard Elements for definitions of the «extends», «extends», and «useCaseModel» stereotypes.

2.12.6 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.

2.13 State Machines

2.13.1 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.

2.13.2 Abstract Syntax

The abstract syntax for the State Machines package is expressed in graphic notation in the following figures. Figure 2-24 on page 2-108 shows the main model elements that define state machines, which include States, Events and Transitions.

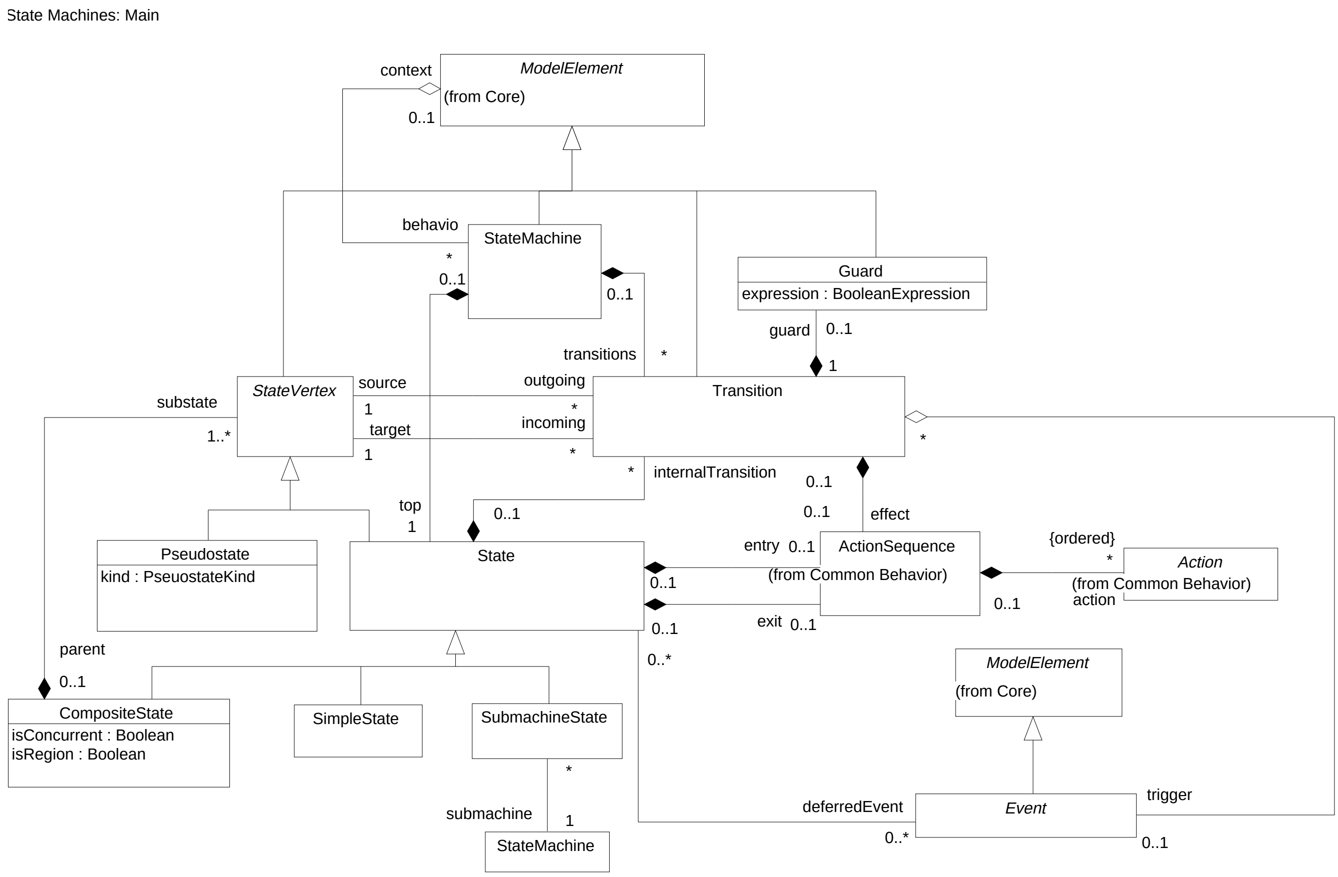


Figure 2-24 State Machines - Main

Figure 2-25 on page 2-109 shows model elements that are specializations of Events.

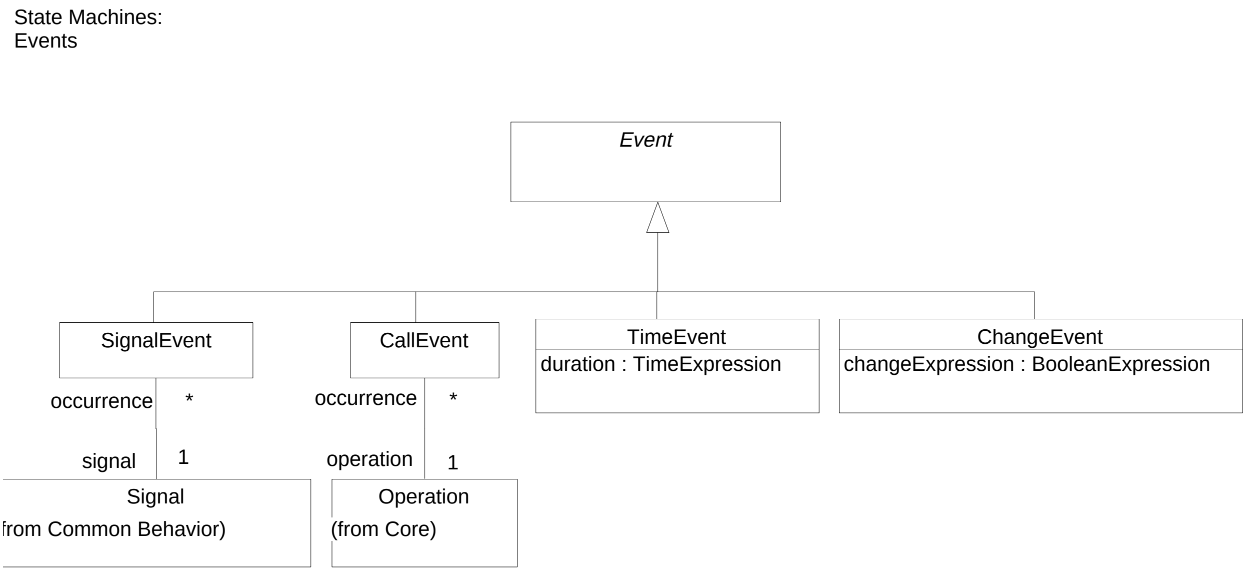


Figure 2-25 State Machines - Events

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

substate	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. 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

kind	Determines the type of the PseudoState and can be one of initial, deepHistory, shallowHistory, join, fork, branch, or 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.

2.13.3 Well-FormednessRules

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))
        ))
```

2.13.4 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 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 – 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 – 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

A state can be active or inactive during execution. 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) 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.
- 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 deep 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).

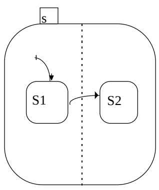
The main source is a direct substate of the LCA that contains the explicit sources. The main target is a substate of the LCA that contains the explicit targets.

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).

2.13.5 Standard Elements

The predefined stereotypes, constraints and tagged values for the State Machines package are listed in Table 2-6 and defined in Appendix A - UML Standard Elements.

Table 2-6 State Machines - Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Event	«create» «destroy»		

2.13.6 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 below.

```
class bankAccount {
private:
int balance;
public;
void deposit (amount)
{
 if (balance > 0) balance = balance + amount’ // no change
 else
 balance = balance + amount - 1; // $1 charge for the trans-
action
}
void withdrawal (amount) {
if (balance>0) balance = balance - amount ;
}
}
```

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 2-26 on page 2-129.

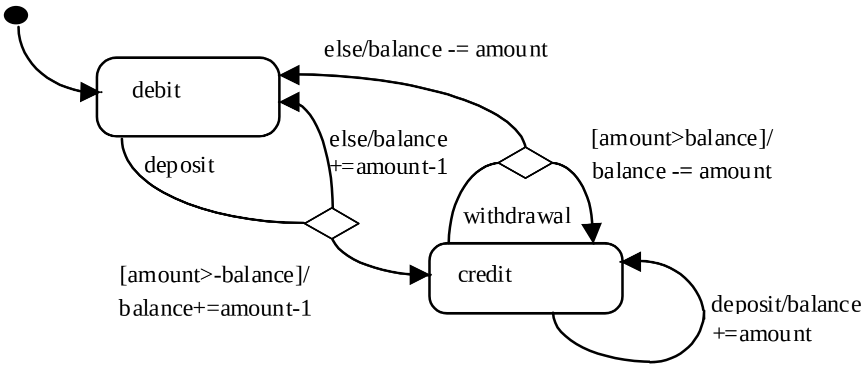


Figure 2-26 State Machine for Modeling Class Behavior

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.’

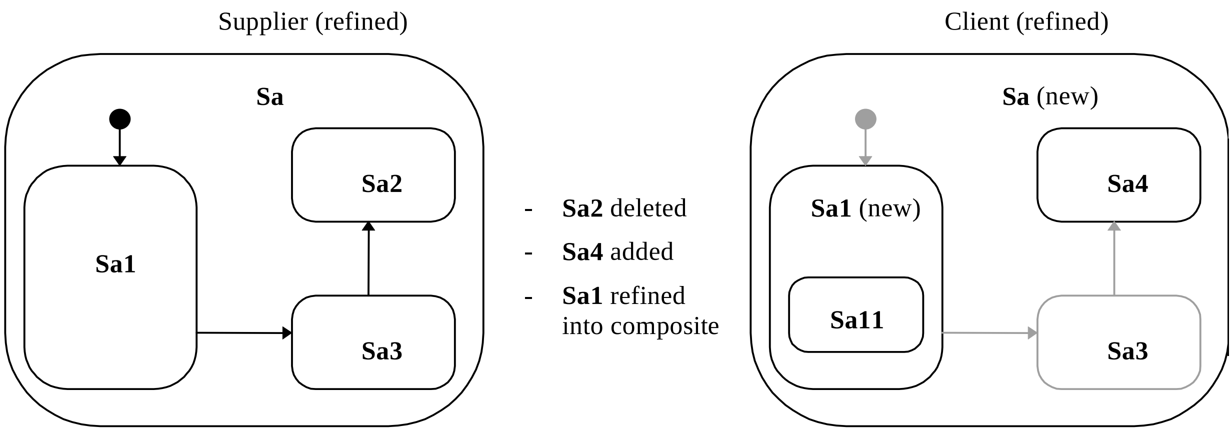


Figure 2-27 State Machine Refinement Example

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, and 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.

2.13.7 Activity Models

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 REVIEWER: PLEASE FINISH THIS SENTENCE**. Transitions into action states are triggered by events, which can be

- the completion of a previous action state,
- the availability of an object in a certain state,
- the occurrence of a signal; or
- 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.

2.13.7.1 Abstract Syntax

The abstract syntax for activity models is expressed in graphic notation in Figure 2-1 on page 2-134.

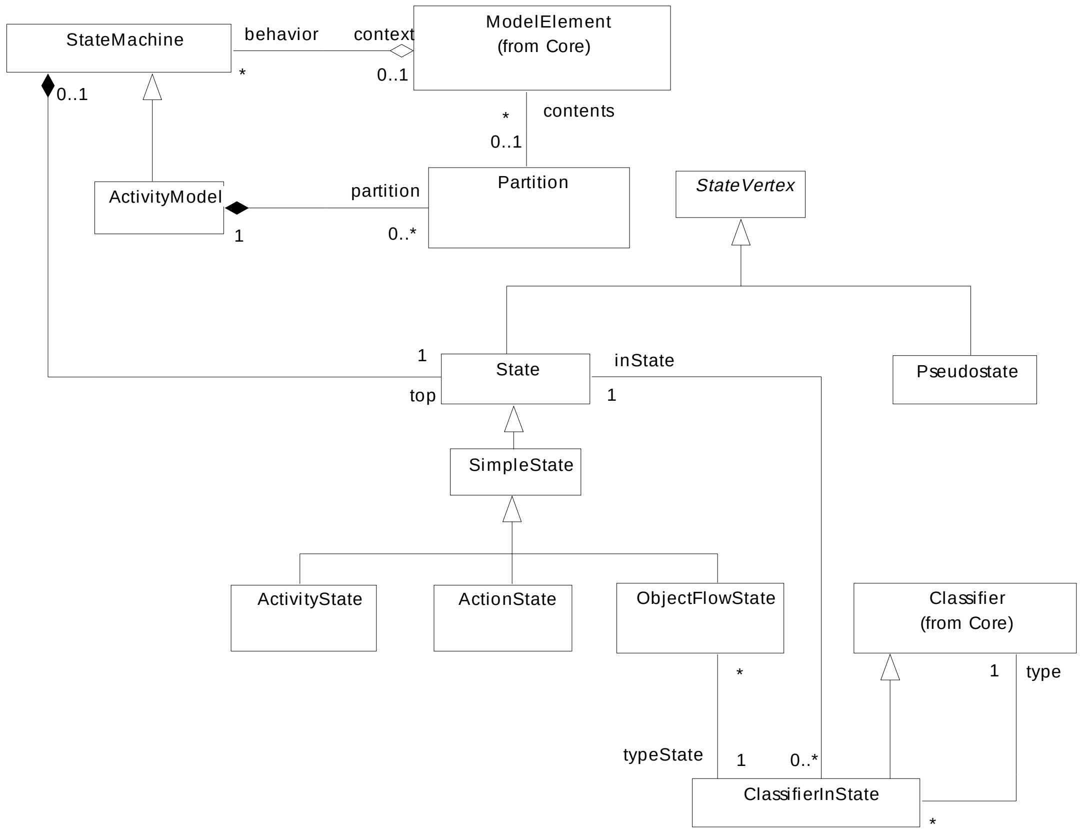


Figure 2-1 Activity Models

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.

2.13.7.2 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.

2.13.7.3 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.

2.13.7.4 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:

1. 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.
2. The ‘contents’ of object flow states are typed. They are not unstructured data definitions as in data stores.
3. 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

2.14 Model Management

This section defines the 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.

2.14.1 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.

2.14.2 Abstract Syntax

The abstract syntax for the Model Management package is expressed in graphic notation in Figure 2-1.

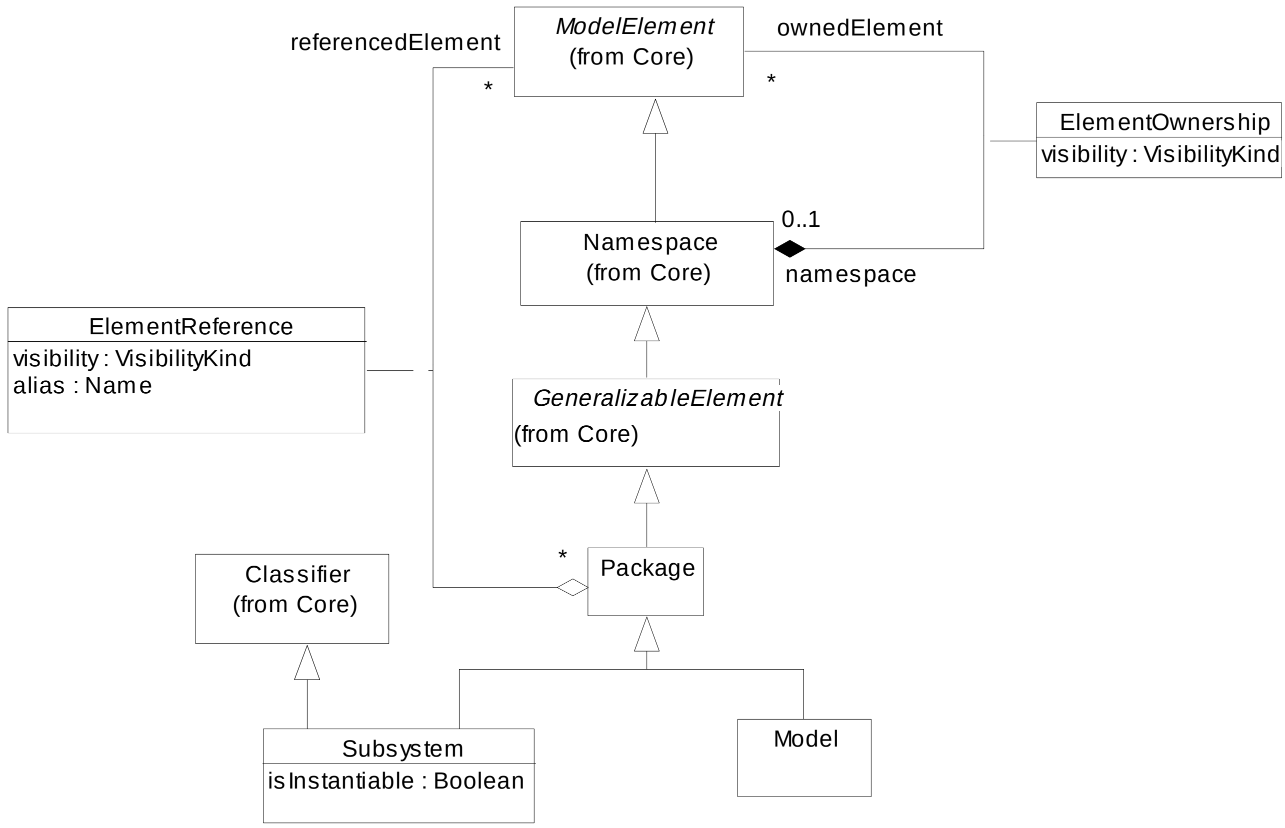


Figure 2-1 Model Management

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: 1) specification elements and 2) 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.

2.14.3 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) )
```

2.14.4 Semantics

Package

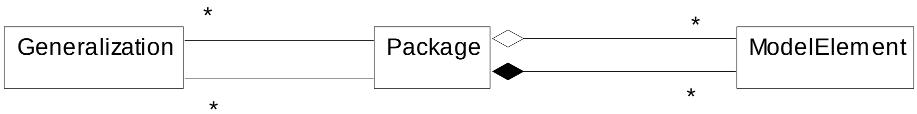


Figure 2-2 Package Illustration

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: 1) by importing their containing packages or 2) by defining generalizations 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; however, 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 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 that 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 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

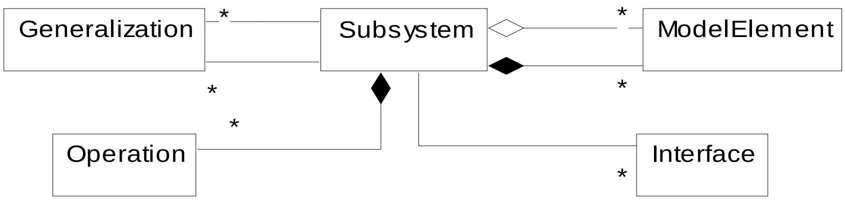


Figure 2-3 Subsystem Illustration

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 have 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: 1) specification elements and 2) 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 which then performs a method.

Importing subsystems is done in the same way as 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 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

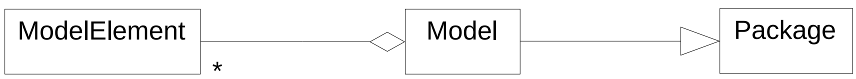


Figure 2-4 Model Illustration

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 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.

2.14.5 Standard Elements

The predefined stereotypes, constraints, and tagged values for the Model Management package are listed in Table 2-7 and defined in Appendix A - UML Standard Elements.

Table 2-7 Model Management - Standard Elements

Model Element	Stereotypes	Constraints	Tagged Values
Model	«useCaseModel»		
Package	«facade» «framework» «stub» «system» «topLevelPackage»		

2.14.6 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 include:

- 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.

UML Notation Guide 3

This guide describes the notation for the visual representation of the Unified Modeling Language (UML). This notation document contains brief summaries of the semantics of UML constructs, but the UML Semantics chapter must be consulted for full details.

Contents

This chapter contains the following topics.

Part 1 - Background

3.1 Introduction

This chapter is arranged in parts according to semantic concepts subdivided by diagram types. Within each diagram type, model elements that are found on that diagram and their representation are listed. Note that many model elements are usable in more than one diagram. An attempt has been made to place each description where it is used the most, but be aware that the document involves implicit cross-references and that elements may be useful in places other than the section in which they are described. Be aware also that the document is nonlinear: there are forward references in it. It is not intended to be a teaching document that can be read linearly, but a reference document organized by affinity of concept.

Each part of this chapter is divided into sections, roughly corresponding to important model elements and notational constructs. Note that some of these constructs are used within other constructs; do not be misled by the flattened structure of the chapter. Within each section the following subsections may be found:

- Semantics: Brief summary of semantics. For a fuller explanation and discussion of fine points, see the *UML Semantics* chapter in this document.
- Notation: Explains the notational representation of the semantic concept (“forward mapping to notation”).
- Presentation options: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of presenting information within a tool.

  Dynamic tools need the freedom to present information in various ways and the authors do not want to restrict this excessively. In some sense, we are defining the “canonical notation” that printed documents show, rather than the “screen notation.” The ability to extend the notation can lead to unintelligible dialects, so we hope this freedom will be used in intuitive ways. The authors have not sought to eliminate all the ambiguity that some of these presentation options may introduce, because the presence of the underlying model in a dynamic tool serves to easily disambiguate things. Note that a tool is not supposed to pick just one of the presentation options and implement it. Tools should offer users the options of selecting among various presentation options, including some that are not described in this document.
- Style guidelines: Include suggestions for the use of stylistic markers, such as fonts, naming conventions, arrangement of symbols, etc., that are not explicitly part of the notation, but that help to make diagrams more readable. These are similar to text indentation rules in C++ or Smalltalk. Not everyone will choose to follow these suggestions, but the use of some consistent guidelines of your own choosing is recommended in any case.
- Example: Shows samples of the notation. String and code examples are given in the following font: `This is a string sample.`

- Mapping: Shows the mapping of notation elements to metamodel elements ("reverse mapping from notation"). This indicates how the notation would be represented as semantic information. Note that, in general, diagrams are interpreted in a particular context in which semantic and graphic information is gathered simultaneously. The assumption is that diagrams are constructed by an editing tool that internalizes the model as the diagram is constructed. Some semantic constructs have no graphic notation and would be shown to a user within a tool using a form or table.

Part 2 - Diagram Elements

3.2 Graphs and Their Contents

Most UML diagrams and some complex symbols are graphs containing nodes connected by paths. The information is mostly in the topology, not in the size or placement of the symbols (there are some exceptions, such as a sequence diagram with a metric time axis). There are three kinds of visual relationships that are important::

1. connection (usually of lines to 2-d shapes),
2. containment (of symbols by 2-d shapes with boundaries), and
3. visual attachment (one symbol being "near" another one on a diagram).

These visual relationships map into connections of nodes in a graph, the parsed form of the notation.

UML notation is intended to be drawn on 2-dimensional surfaces. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes), but they are still rendered as icons on a 2-dimensional surface. In the near future, true 3-dimensional layout and navigation may be possible on desktop machines; however, it is not currently practical.

There are basically four kinds of graphical constructs that are used in UML notation:

1. Icons - An icon is a graphical figure of a fixed size and shape. It does not expand to hold contents. Icons may appear within area symbols, as terminators on paths or as standalone symbols that may or may not be connected to paths.

2. 2-d Symbols - Two-dimensional symbols have variable height and width and they can expand to hold other things, such as lists of strings or other symbols. Many of them are divided into compartments of similar or different kinds. Paths are connected to two-dimensional symbols by terminating the path on the boundary of the symbol. Dragging or deleting a 2-d symbol affects its contents and any paths connected to it.

3. Paths - Sequences of line segments whose endpoints are attached. Conceptually a path is a single topological entity, although its segments may be manipulated graphically. A segment may not exist apart from its path. Paths are always attached

to other graphic symbols at both ends (no dangling lines). Paths may have *terminators*, that is, icons that appear in some sequence on the end of the path and that qualify the meaning of the path symbol.

4. Strings - Present various kinds of information in an “unparsed” form. UML assumes that each usage of a string in the notation has a syntax by which it can be parsed into underlying model information. For example, syntaxes are given for attributes, operations, and transitions. These syntaxes are subject to extension by tools as a presentation option. Strings may exist as singular elements of symbols or compartments of symbols, as elements in lists (in which case the position in the list conveys information), as labels attached to symbols or paths, or as stand-alone elements on a diagram.

3.3 Drawing Paths

A path consists of a series of line segments whose endpoints coincide. The entire path is a single topological unit. Line segments may be orthogonal lines, oblique lines, or curved lines. Certain common styles of drawing lines exist: all orthogonal lines, or all straight lines, or curves only for bevels. The line style can be regarded as a tool restriction on default line input. When line segments cross, it may be difficult to know which visual piece goes with which other piece; therefore, a crossing may optionally be shown with a small semicircular jog by one of the segments to indicate that the paths do not intersect or connect (as in an electrical circuit diagram).

In some relationships (such as aggregation and generalization) several paths of the same kind may connect to a single symbol. In some circumstances (described for the particular relationship) the line segments connected to the symbol can be combined into a single line segment, so that the path from that symbol branches into several paths in a kind of tree. This is purely a graphical presentation option; conceptually the individual paths are distinct. This presentation option may not be used when the modeling information on the segments to be combined is not identical.

3.4 Invisible Hyperlinks and the Role of Tools

A notation on a piece of paper contains no hidden information. A notation on a computer screen may contain additional invisible hyperlinks that are not apparent in a static view, but that can be invoked dynamically to access some other piece of information, either in a graphical view or in a textual table. Such dynamic links are as much a part of a *dynamic* notation as the visible information, but this guide does not prescribe their form. We regard them as a tool responsibility. This document attempts to define a *static* notation for the UML, with the understanding that some useful and interesting information may show up poorly or not at all in such a view. On the other hand, we do not know enough to specify the behavior of all dynamic tools, nor do we want to stifle innovation in new forms of dynamic presentation. Eventually some of the dynamic notations may become well enough established to standardize them, but we do not feel that we should do so now.

3.5 Background Information

3.5.1 Presentation Options

Each appearance of a symbol for a class on a diagram or on different diagrams may have its own presentation choices. For example, one symbol for a class may show the attributes and operations and another symbol for the same class may suppress them. Tools may provide style sheets attached either to individual symbols or to entire diagrams. The style sheets would specify the presentation choices. (Style sheets would be applicable to most kinds of symbols, not just classes.)

Not all modeling information is presented most usefully in a graphical notation. Some information is best presented in a textual or tabular format. For example, much detailed programming information is best presented as text lists. The UML does not assume that all of the information in a model will be expressed as diagrams; some of it may only be available as tables. This document does not attempt to prescribe the format of such tables or of the forms that are used to access them, because the underlying information is adequately described in the UML metamodel and the responsibility for presenting tabular information is a tool responsibility. It is assumed that hidden links may exist from graphical items to tabular items.

3.6 String

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

3.6.1 Semantics

Diagram strings normally map underlying model strings that store or encode information about the model, although some strings may exist purely on the diagrams. UML assumes that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that the tool and the computer manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used without further fuss.

3.6.2 Notation

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent. Depending on purpose, a string might be shown as a single-line entity or as a paragraph with automatic line breaks.

Typeface and font size are graphic markers that are normally independent of the string itself. They may code for various model properties, some of which are suggested in this document and some of which are left open for the tool or the user.

3.6.3 Presentation Options

Tools may present long strings in various ways, such as truncation to a fixed size, automatic wrapping, or insertion of scroll bars. It is assumed that there is a way to obtain the full string dynamically.

3.6.4 Example

BankAccount

integrate (f: Function, from: Real, to: Real)

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

The purpose of the shuffle operation is nominally to put the cards into a random configuration. However, to more closely capture the behavior of physical decks, in which blocks of cards may stick together during several riffles, the operation is actually simulated by cutting the deck and merging the cards with an imperfect merge.

3.6.5 Mapping

A graphic string maps into a string within a model element. The mapping depends on context. In some circumstances, the visual string is parsed into multiple model elements. For example, an operation signature is parsed into its various fields. Further details are given with each kind of symbol.

3.7 Name

3.7.1 Semantics

A name is a string that is used to identify a model element uniquely within some scope. A pathname is used to find a model element starting from the root of the system (or from some other point). A name is a selector (qualifier) within some scope—the scope is made clear in this document for each element that can be named.

A pathname is a series of names linked together by a delimiter (such as ‘::’). There are various kinds of pathnames described in this document, each in its proper place and with its particular delimiter.

3.7.2 Notation

A name is displayed as a text string graphic. Normally a name is displayed on a single line and will not contain nonprintable characters. Tools and languages may impose reasonable limits on the length of strings and the character set they use for names, possibly more restrictive than those for arbitrary strings, such as comments.

3.7.3 Example

Names:

BankAccount

integrate

controller

abstract

this_is_a_very_long_name_with_underscores

Pathname:

MathPak::Matrices::BandedMatrix.dimension

3.7.4 Mapping

Maps to the name of a model element. The mapping depends on context, as with String. Further details are given with the particular element.

3.8 Label

A label is a string that is attached to a graphic symbol.

3.8.1 Semantics

A label is a term for a particular use of a string on a diagram. It is purely a notational term.

3.8.2 Notation

A label is a string that is attached graphically to another symbol on a diagram. Visually the attachment normally is by containment of the string (in a closed region) or by placing the string near the symbol. Sometimes the string is placed in a definite position (such as below a symbol) but most of the time the statement is that the string must be "near" the symbol. A tool maintains an explicit internal graphic linking between a label and a graphic symbol, so that the label drags with the symbol, but the final appearance of the diagram is a matter of aesthetic judgment and should be made so that there is no confusion about which symbol a label is attached to. Although the attachment may not be obvious from a visual inspection of a diagram, the attachment is clear and unambiguous at the graphic level (and poses no ambiguity in the semantic mapping).

3.8.3 Presentation Options

A tool may visually show the attachment of a label to another symbol using various aids (such as a line in a given color, flashing of matched elements, etc.) as a convenience.

3.8.4 Example

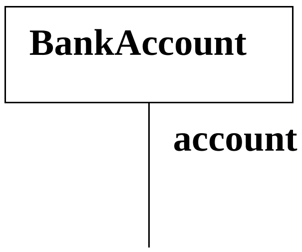


Figure 3-1 Attachment by Containment and Attachment by Adjacency

3.9 Keywords

The number of easily-distinguishable visual symbols is limited. The UML notation makes use of text keywords in places to distinguish variations on a common theme, including metamodel subclasses of a base class, stereotypes of a metamodel base class, and groups of list elements. From the user's perspective, the metamodel distinction between metamodel subclasses and stereotypes is often unimportant, although it is important to tool builders and others who implement the metamodel.

The general notation for the use of a keyword is to enclose it in guillemets («»):

«*keyword***»**

Certain predefined keywords are described in the text of this document. These must be treated as reserved words in the notation. Others are available for users to employ as stereotype names. The use of a stereotype name that matches a predefined keyword is ill-formed.

3.10 Expression

3.10.1 Semantics

Various UML constructs require *expressions*, which are linguistic formulas that yield values when evaluated at run-time. These include expressions for types, boolean values, and numbers. UML does not include an explicit linguistic analyzer for expressions. Rather, expressions are expressed as strings in a particular *language*. The

OCL constraint language is used within the UML semantic definition and may also be used at the user level; other languages (such as programming languages) may also be used.

UML avoids specifying the syntax for constructing type expressions because they are so language-dependent. It is assumed that the name of a class or simple data type will map into a simple *Classifier* reference, but the syntax of complicated language-dependent type expressions, such as C++ function pointers, is the responsibility of the specification language.

3.10.2 Notation

An expression is displayed as a string defined in a particular language. The syntax of the string is the responsibility of a tool and a linguistic analyzer for the language. The assumption is that the analyzer can evaluate strings at run-time to yield values of the appropriate type, or can yield semantic structures to capture the meaning of the expression. For example, a type expression evaluates to a Classifier reference, and a boolean expression evaluates to a true or false value. The language itself is known to a modeling tool but is generally implicit on the diagram, under the assumption that the form of the expression makes its purpose clear.

3.10.3 Example

BankAccount

BankAccount * (*) (Person*, int)

array [1..20] of reference to range (-1.0..1.0) of Real

[i > j and self.size > i]

3.10.4 Mapping

An expression string maps to an Expression element (possibly a particular subclass of Expression, such as ObjectSetExpression or TimeExpression).

3.10.5 OCL Expressions

UML includes a definition of the OCL language, which is used to define constraints within the UML metamodel itself. The OCL language may be supported by tools for user-written expressions as well. Other possible languages include various computer languages as well as plain text (which cannot be parsed by a tool, of course, and is therefore only for human information).

3.10.6 Selected OCL Notation

Syntax for some common navigational expressions are shown below. These forms can be chained together. The leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable. For more details and syntax see the OCL description.

item '.' *selector*	the *selector* is the name of an attribute in the item or the name of a role of the target end of a link attached to the item. The result is the value of the attribute or the related object(s). The result is a value or a set of values depending on the multiplicities of the item and the association.
item '.' *selector* '[' *qualifier-value* ']'	the *selector* designates a qualified association that qualifies the *item*. The *qualifier-value* is a value for the qualifier attribute. The result is the related object selected by the qualifier. Note that this syntax is applicable to array indexing as a form of qualification.
set '->' 'select' '(' *boolean-expression* ')'	the *boolean-expression* is written in terms of objects within the set. The result is the subset of objects in the set for which the boolean expression is true.

3.10.7 Example

```
flight.pilot.training_hours > flight.plane.minimum_hours

company.employees–>select (title = “Manager” and self.reports–>size > 10)
```

3.11 Note

A note is a graphical symbol containing textual information (possibly including embedded images). It is a notation for rendering various kinds of textual information from the metamodel, such as constraints, comments, method bodies, and tagged values.

3.11.1 Semantics

A note is a notational item. It shows textual information within some semantic element.

3.11.2 Notation

A note is shown as a rectangle with a “bent corner” in the upper right corner. It contains arbitrary text. It appears on a particular diagram and may be attached to zero or more modeling elements by dashed lines.

3.11.3 Presentation Options

A note may have a stereotype.

A note with the stereotype “constraint” or a more specific form of constraint (such as the code body for a method) designates a constraint that is part of the model and not just part of a diagram view. Such a note is the view of a model element (the constraint). Other kinds of notes are purely notation, they have no underlying model element.

3.11.4 Example

See also Figure 3-5 on page 3-20 for a note symbol containing a constraint.

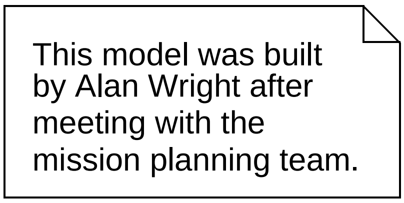


Figure 3-2 Note

3.11.5 Mapping

A note may represent the textual information in several possible metamodel constructs; it must be created in context that is known to a tool, and the tool must maintain the mapping. The string in the note maps to the body of the corresponding modeling element. A note may represent:

- a constraint,
- a tagged value,
- the body of a method, or
- other string values within modeling elements.

It may also represent a comment attached directly to a diagram element.

3.12 Type-Instance Correspondence

A major purpose of modeling is to prepare generic descriptions that describe many specific items. This is often known as the *type-instance dichotomy.* Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, Parameter-Value, Operation-Call, and so on.

Although diagrams for type-like elements and instance-like elements are not exactly the same, they share many similarities. Therefore, it is convenient to choose notation for each type-instance pair of elements such that the correspondence is visually apparent immediately. There are a limited number of ways to do this, each with advantages and disadvantages. In UML, the type-instance distinction is shown by employing the same geometrical symbol for each pair of elements and by underlining the name string (including type name, if present) of an instance element. This visual distinction is generally easily apparent without being overpowering even when an entire diagram contains instance elements.

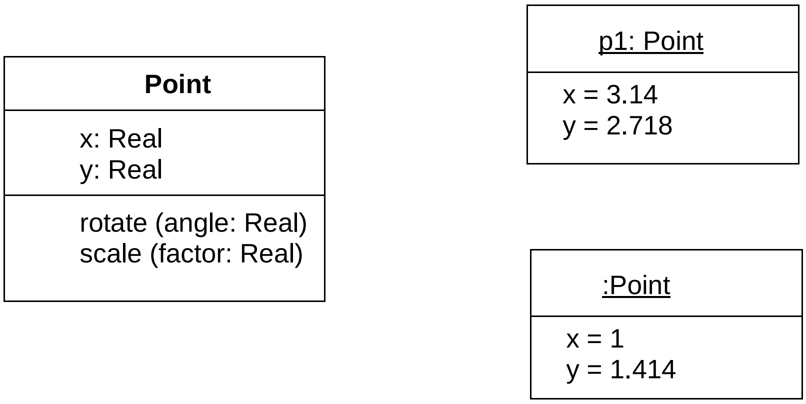


Figure 3-3 Classes and Objects

A tool is free to substitute a different graphic marker for instance elements at the user's option, such as color, fill patterns, or so on.

Part 3 - Model Management

3.13 Packages and Model Organization

3.13.1 Semantics

A *package* is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain both subordinate packages and ordinary model elements. Some packages may be Subsystems or Models. The entire system description can be thought of as a single high-level subsystem package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages.

Note that packages *own* model elements and model fragments and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree. However, packages can reference other packages, so the usage network is a graph.

There are several predefined stereotypes of Model and Subsystem. See the Meta Object Facility (MOF) Specification for details. In particular, the stereotype «system» of Subsystem denotes the entire set of models for the complete system being modeled. It is the root of the package hierarchy and the only model element that is not owned by some other model element.

3.13.2 Notation

A package is shown as a large rectangle with a small rectangle (a "tab") attached on one corner (usually the left side of the upper side of the large rectangle). It is a manila folder shape.

- If contents of the package are not shown, then the name of the package is placed within the large rectangle.
- If contents of the package are shown, then the name of the package may be placed within the tab.

A keyword string may be placed above the package name. The keywords *subsystem* and *model* indicate that the package is a metamodel Subsystem or Model. The predefined stereotypes *system, facade, framework,* and *top package* are also notated with keywords. User-defined stereotypes of one of these predefined kinds of package are also notated with keywords, but they must not conflict with the predefined keywords.

A list of properties may be placed in braces after or below the package name. Example: {abstract}. See Section 3.15, "Element Properties," on page 3-21 for details of property syntax.

The contents of the package may be shown within the large rectangle.

The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol ('+' for public, '-' for private, '#' for protected). If the element is an inner package, the visibilities of its elements, as exported by the outer package, are obtained by:

- combining the visibilities of an element within the package, with
- the visibility of the package itself.

The most restrictive visibility results. Relationships may be drawn between package symbols to show relationships between some of the elements in the packages. In particular, dependency between packages implies that one or more dependencies among the elements exists.

3.13.3 Presentation Options

A tool may also show visibility by selectively displaying those elements that meet a given visibility level (e.g., all of the public elements only).

A tool may show visibility by a graphic marker, such as color or font.

3.13.4 Style Guidelines

It is expected that packages with large contents will be shown as simple icons with names, in which the contents may be dynamically accessed by “zooming” to a detailed view.

3.13.5 Example

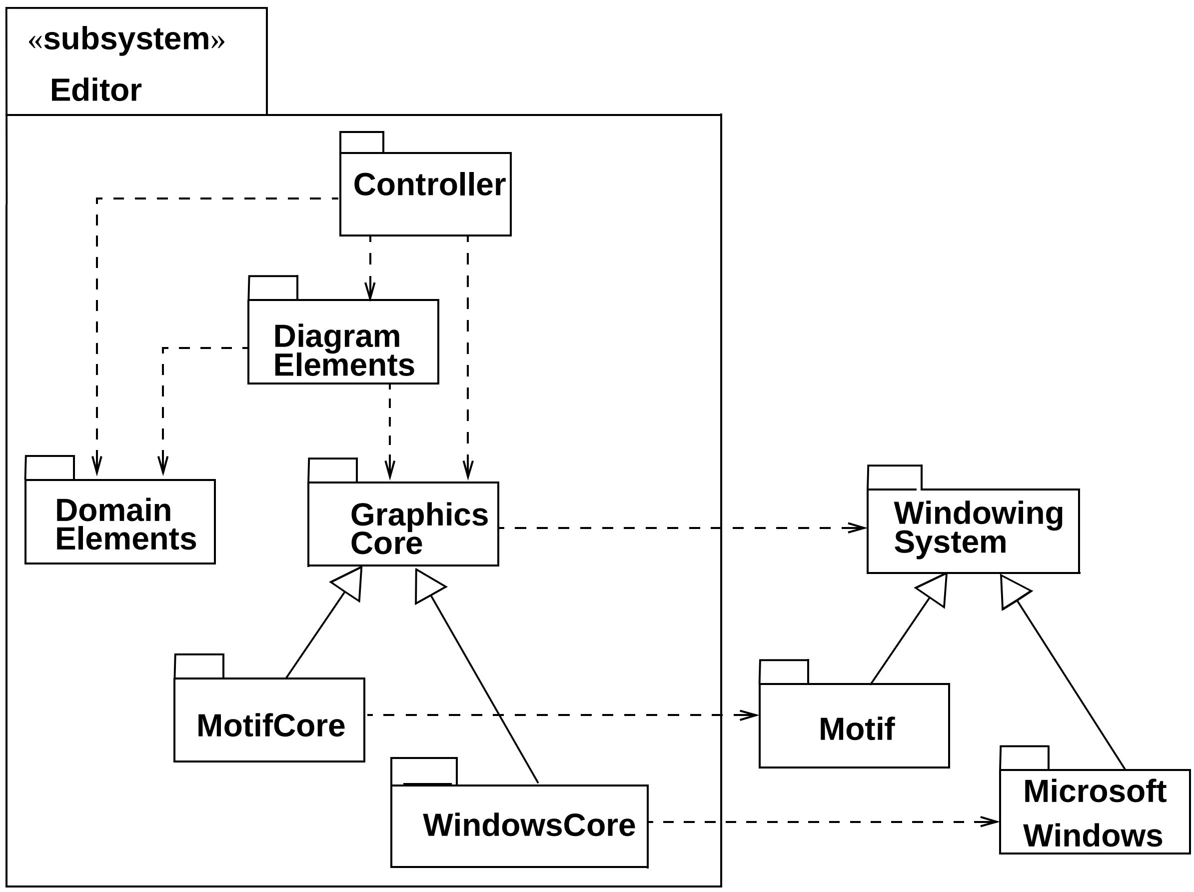


Figure 3-4 Packages and Their Dependencies

3.13.6 Mapping

A package symbol maps into a Package element. The name on the package symbol is the name of the Package element. If the package has a keyword that is a predefined keyword, then the package symbol maps into the corresponding subclass of Package or into the corresponding stereotype of Package; otherwise, it maps into a user-defined stereotype of Package.

A symbol directly contained within the package symbol (i.e., not contained within another symbol) maps into a model element owned by the package element. However, a symbol whose name is a pathname maps into a reference to a model element owned by another package. Only the reference is owned by the current package. Relationships from the package symbol boundary map into relationships to the package element.

Part 4 - General Extension Mechanisms

The elements in this section are general purpose mechanisms that may be applied to any modeling element. The semantics of a particular use depends on a convention of the user or an interpretation by a particular constraint language or programming language; therefore, they constitute an extensibility device for UML.

3.14 Constraint and Comment

3.14.1 Semantics

A *constraint* is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true; otherwise, the system described by the model is invalid (with consequences that are outside the scope of UML). Certain kinds of constraints (such as an association “or” constraint) are predefined in UML, others may be user-defined. A user-defined constraint is described in words in a given language, whose syntax and interpretation is a tool responsibility. A constraint represents semantic information attached to a model element, not just to a view of it.

A *comment* is a text string (including references to human-readable documents) attached directly to a model element. This is equivalent syntactically to a constraint written in the language “text” whose meaning is significant to humans, but which is not conceptually executable (except inasmuch as humans are regarded as the instruments of interpretation). A comment can attach arbitrary textual information to any model element of presumed general importance.

3.14.2 Notation

A constraint is shown as a text string in braces ({ }). There is an expectation that individual tools may provide one or more languages in which formal constraints may be written. One predefined language for writing constraints is OCL (see Object

Constraint Language Specification, chapter 4); otherwise, the constraint may be written in natural language. A constraint may be a “comment.” In that case, it is written in text (possibly including pictures or other viewable documents) for “interpretation” by a human. Each constraint is written in a specific language, although the language is not generally displayed on the diagram (the tool must keep track of it).

For an element whose notation is a text string (such as an attribute, etc.), the constraint string may follow the element text string in braces.

For a list of elements whose notation is a list of text strings (such as the attributes within a class), a constraint string may appear as an element in the list. The constraint applies to all succeeding elements of the list until another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraint, but may augment or modify individual constraints within the constraint string.

For a single graphical symbol (such as a class or an association path), the constraint string may be placed near the symbol, preferably near the name of the symbol, if any.

For two graphical symbols (such as two classes or two associations), the constraint is shown as a dashed arrow from one element to the other element labeled by the constraint string (in braces). The direction of the arrow is relevant information within the constraint.

For three or more graphical symbols, the constraint string is placed in a note symbol and attached to each of the symbols by a dashed line. This notation may also be used for the other cases. For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.

A comment is shown by a text string placed within a note symbol that is attached to a model element. The braces are omitted to show that this is purely a textual comment. (The braces indicate a constraint expressed in some interpretable constraint language.)

3.14.3 Example

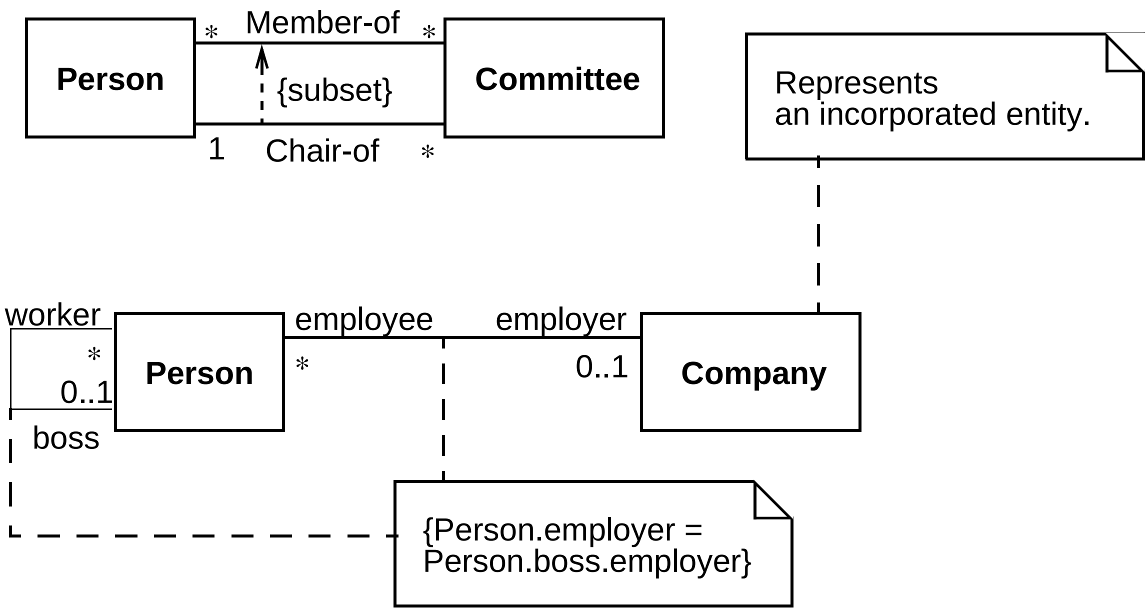


Figure 3-5 Constraints Illustration

3.14.4 Mapping

The constraint string maps into the *body* expression in a Constraint element. The mapping depends on the language of the expression, which is known to a tool but generally not displayed on a diagram. If the string lacks braces (i.e., a Comment), then it maps into an expression in the language “text.”

A constraint string following a list entry maps into a Constraint attached to the element corresponding to the list entry.

A constraint string represented as a stand-alone list element maps into a separate Constraint attached to each succeeding model element corresponding to subsequent list entries (until superseded by another constraint or property string).

A constraint string placed near a graphical symbol must be attached to the symbol by a hidden link by a tool operating in context. The tool must maintain the graphical linkage implicitly. The constraint string maps into a Constraint attached to the element corresponding to the symbol.

A constraint string attached to a dashed arrow maps into a constraint attached to the two elements corresponding to the symbols connected by the arrow.

A constraint string in a note symbol maps into a Constraint attached to the elements corresponding to the symbols connected to the note symbol by dashed lines.

3.15 Element Properties

Many kinds of elements have detailed properties that do not have a visual notation. In addition, users can define new element properties using the *tagged value* mechanism.

A string may be used to display properties attached to a model element. This includes properties represented by attributes in the metamodel as well as both predefined and user-defined tagged values.

3.15.1 Semantics

Note that we use *property* in a general sense to mean any value attached to a model element, including attributes, associations, and tagged values. In this sense it can include indirectly reachable values that can be found starting at a given element.

A *tagged value* is a keyword-value pair that may be attached to any kind of model element (including diagram elements as well as semantic model elements). The keyword is called a *tag*. Each tag represents a particular kind of property applicable to one or many kinds of model elements. Both the tag and the value are encoded as strings. Tagged values are an extensibility mechanism of UML permitting arbitrary information to be attached to models. It is expected that most model editors will provide basic facilities for defining, displaying, and searching tagged values as strings but will not otherwise use them to extend the UML semantics. It is expected, however, that back-end tools such as code generators, report writers, and the like will read tagged values to alter their semantics in flexible ways.

3.15.2 Notation

A property (either a metamodel attribute or a tagged value) is displayed as a comma-delimited sequence of *property specifications* all inside a pair of braces ({ }).

A *property specification* has the form

keyword = value

where *keyword* is the name of a property (metamodel attribute or arbitrary tag) and *value* is an arbitrary string that denotes its value. If the type of the property is Boolean, then the default value is **true** if the value is omitted. That is, to specify a value of true you may include just the keyword. To specify a value of false, you omit the name completely. Properties of other types require explicit values. The syntax for displaying the value is a tool responsibility in cases where the underlying model value is not a string or a number.

Note that property strings may be used to display built-in attributes as well as tagged values.

3.15.3 Presentation Options

A tool may present property specifications on separate lines with or without the enclosing braces, provided they are marked appropriately to distinguish them from other information. For example, properties for a class might be listed under the class name in a distinctive typeface, such as italics or a different font family.

3.15.4 Style Guidelines

It is legal to use strings to specify properties that have graphical notations; however, such usage may be confusing and should be used with care.

3.15.5 Example

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

{ abstract }

3.15.6 Mapping

Each term within a string maps to either a built-in attribute of a model element or a tagged value (predefined or user-defined). A tool must enforce the correspondence to built-in attributes.

3.16 Stereotypes

3.16.1 Semantics

A stereotype is, in effect, a new class of modeling element that is introduced at modeling time. It represents a subclass of an existing modeling element with the same form (attributes and relationships) but with a different intent. Generally a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base class. It is expected that code generators and other tools will treat stereotyped elements specially. Stereotypes represent one of the built-in extensibility mechanisms of UML.

3.16.2 Notation

The general presentation of a stereotype is to use the symbol for the base element but to place a keyword string above the name of the element (if any). The keyword string is the name of the stereotype within matched *guillemets,* which are the quotation mark symbols used in French and certain other languages (for example, «foo»).

Note – A guillemet looks like a double angle-bracket, but it is a single character in most extended fonts. Most computers have a Character Map utility. Double angle-brackets may be used as a substitute by the typographically challenged.

The keyword string is generally placed above or in front of the name of the model element being described. The keyword string may also be used as an element in a list, in which case it applies to subsequent list elements until another stereotype string replaces it, or an empty stereotype string («») nullifies it. Note that a stereotype name should not be identical to a predefined keyword applicable to the same element type.

To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype. The UML does not specify the form of the graphic specification, but many bitmap and stroked formats exist (and their portability is a difficult problem). The icon can be used in one of two ways:

1. It may be used instead of, or in addition to, the stereotype keyword string as part of the symbol for the base model element that the stereotype is based on. For example, in a class rectangle it is placed in the upper right corner of the name compartment. In this form, the normal contents of the item can be seen.

2. The entire base model element symbol may be “collapsed” into an icon containing the element name or with the name above or below the icon. Other information contained by the base model element symbol is suppressed. More general forms of icon specification and substitution are conceivable, but we leave these to the ingenuity of tool builders, with the warning that excessive use of extensibility capabilities may lead to loss of portability among tools.

UML avoids the use of graphic markers, such as color, that present challenges for certain persons (the color blind) and for important kinds of equipment (such as printers, copiers, and fax machines). None of the UML symbols *require* the use of such graphic markers. Users *may* use graphic markers freely in their personal work for their own purposes (such as for highlighting within a tool) but should be aware of their limitations for interchange and be prepared to use the canonical forms when necessary.

The classification hierarchy of the stereotypes themselves could be displayed on a class diagram; however, this would be a metamodel diagram and must be distinguished (by user and tool) from an ordinary model diagram. In such a diagram each stereotype is shown as a class with the stereotype «stereotype» (yes, this is a self-referential usage). Generalization relationships may show the extended metamodel hierarchy. Because of the danger of extending the internal metamodel hierarchy, a tool may, but need not, expose this capability on class diagrams. This is not a capability required by ordinary modelers.

3.16.3 Example

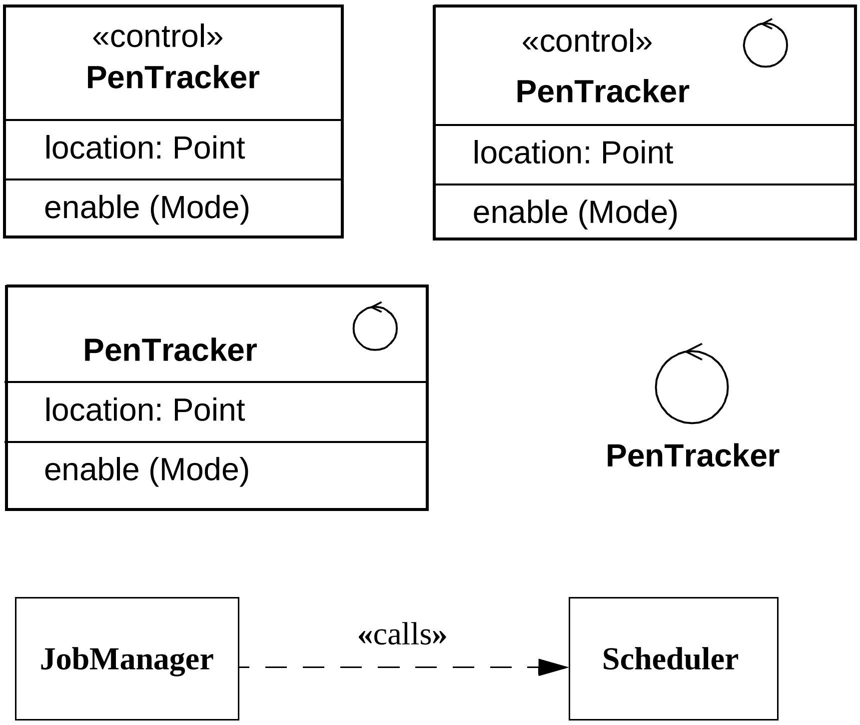


Figure 3-6 Varieties of Stereotype Notation

3.16.4 Mapping

The use of a stereotype keyword maps into the stereotype relationship between the Element corresponding to the symbol containing the name and the Stereotype of the given name. The use of a stereotype icon within a symbol maps into the stereotype relationship between the Element corresponding to the symbol containing the icon and the Stereotype represented by the symbol. A tool must establish the connection when the symbol is created and there is no requirement that an icon represent uniquely one stereotype. The use of a stereotype icon, instead of a symbol, must be created in a context in which a tool implies a corresponding model element and a Stereotype represented by the icon. The element and the stereotype have the stereotype relationship.

Part 5 - Static Structure Diagrams

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. Class diagrams do not show temporal information, although they may contain reified occurrences of things that have or things that describe temporal behavior. An object diagram shows instances compatible with a particular class diagram.

REVIEWER: Instead of using 'thing' in the above paragraph, can we use a different word?

This section discusses classes and their variations, including templates and instantiated classes, and the relationships between classes (association and generalization) and the contents of classes (attributes and operations).

3.17 Class Diagram

A class diagram is a graph of Classifier elements connected by their various static relationships. Note that a "class" diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structural diagram" but "class diagram" is shorter and well established.

3.17.1 Semantics

A class diagram is a graphic view of the static structural model. The individual class diagrams do not represent divisions in the underlying model.

3.17.2 Notation

A class diagram is a collection of (static) declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents. Class diagrams may be organized into packages either with their underlying models or as separate packages that build upon the underlying model packages.

3.17.3 Mapping

A class diagram does not necessarily match a single semantic entity. A package within the static structural model may be represented by one or more class diagrams. The division of the presentation into separate diagrams is for graphical convenience and does not imply a partitioning of the model itself. The contents of a diagram map into elements in the static semantic model. If a diagram is part of a package, then its contents map into elements in the same package.

3.18 Object Diagram

An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, mainly to show examples of data structures.

Tools need not support a separate format for object diagrams. Class diagrams can contain objects, so a class diagram with objects and no classes is an “object diagram.” The phrase is useful, however, to characterize a particular usage achievable in various ways.

3.19 Classifier

Classifier is the metamodel superclass of *Class, DataType,* and *Interface.* All of these have similar syntax and are therefore all notated using the rectangle symbol with keywords used as necessary. Because classes are most common in diagrams, a rectangle without a keyword represents a class, and the other subclasses of *Classifier* are indicated with keywords. In the sections that follow, the discussion will focus on *Class,* but most of the notation applies to the other element kinds as semantically appropriate and as described later under their own sections.

3.20 Class

A *class* is the descriptor for a set of objects with similar structure, behavior, and relationships. UML provides notation for declaring classes and specifying their properties, as well as using classes in various ways. Some modeling elements that are similar in form to classes (such as interfaces, signals, or utilities) are notated using keywords on class symbols; some of these are separate metamodel classes and some are stereotypes of Class. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements.

3.20.1 Semantics

A class represents a concept within the system being modeled. Classes have data structure and behavior and relationships to other elements.

The name of a class has scope within the package in which it is declared and the name must be unique (among class names) within its package.

3.20.2 Basic Notation

A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations.

See “Name Compartment” on page 3-28 and “List Compartment” on page 3-29 for more details.

References

By default a class shown within a package is assumed to be defined within that package. To show a reference to a class defined in another package, use the syntax

Package-name::*Class-name*

as the name string in the name compartment. Compartment names can be used to remove ambiguity, if necessary (“List Compartment” on page 3-29). A full pathname can be specified by chaining together package names separated by double colons (::).

3.20.3 Presentation Options

Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it.

Additional compartments may be supplied as a tool extension to show other predefined or user-defined model properties (for example, to show business rules, responsibilities, variations, events handled, exceptions raised, and so on). Most compartments are simply lists of strings. More complicated formats are possible, but UML does not specify such formats; they are a tool responsibility. Appearance of each compartment should preferably be implicit based on its contents. Compartment names may be used, if needed.

Tools may provide other ways to show class references and to distinguish them from class declarations.

A class symbol with a stereotype icon may be “collapsed” to show just the stereotype icon, with the name of the class either inside the class or below the icon. Other contents of the class are suppressed.

3.20.4 Style Guidelines

(Note that these are recommendations, not mandates.)

- Center class name in boldface.
- Center stereotype name in plain face within guillemets above class name.
- Being class names with an uppercase letter.
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show the names of abstract classes or the signatures of abstract operations in italics.

As a tool extension, boldface may be used for marking special list elements (for example, to designate candidate keys in a database design). This might encode some design property modeled as a tagged value, for example.

Show full attributes and operations when needed and suppress them in other contexts or references.

3.20.5 Example

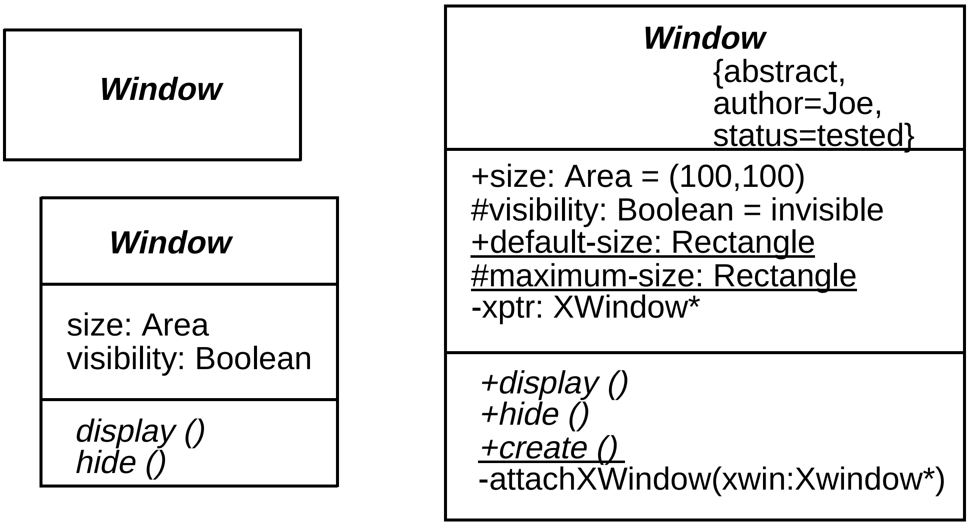


Figure 3-7 Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

3.20.6 Mapping

A class symbol maps into a Class element within the package that owns the diagram. The name compartment contents map into the class name and into properties of the class (built-in attributes or tagged values). The attribute compartment maps into a list of Attributes of the Class. The operation compartment maps into a list of Operations of the Class.

3.21 Name Compartment

3.21.1 Notation

Displays the name of the class and other properties in up to three sections:

An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment. The stereotype name must not match a predefined keyword.

The name of the class appears next. If the class is abstract, its name appears in italics. Note that any explicit specification of generalization status takes precedence over the name font.

A list of strings denoting properties (metamodel attributes or tagged values) may be placed in braces below the class name. The list may show class-level attributes for which there is no UML notation and it may also show tagged values. The presence of a keyword for a Boolean type without a value implies the value *true*. For example, a leaf class shows the property “{leaf}”.

The stereotype and property list are optional.

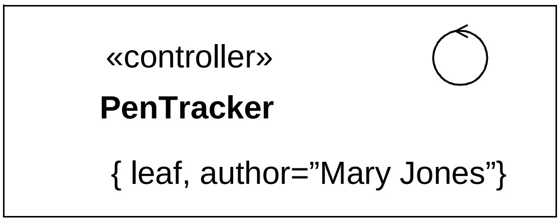


Figure 3-8 Name Compartment

3.21.2 Mapping

The contents of the name compartment map into the name, stereotype, and various properties of the Class represented by the class symbol.

3.22 List Compartment

3.22.1 Notation

Holds a list of strings, each of which is the encoded representation of a feature, such as an attribute or operation. The strings are presented one to a line with overflow to be handled in a tool-dependent manner. In addition to lists of attributes or operations, optional lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. UML does not define these optional lists. The manipulation of user-defined lists is tool-dependent.

The items in the list are ordered and the order may be modified by the user. The order of the elements is meaningful information and must be accessible within tools (for example, it may be used by a code generator in generating a list of declarations). The list elements may be presented in a different order to achieve some other purpose (for example, they may be sorted in some way). Even if the list is sorted, the items maintain their original order in the underlying model. The ordering information is merely suppressed in the view.

An ellipsis (. . .) as the final element of a list or the final element of a delimited section of a list indicates that additional elements in the model exist that meet the selection condition, but that are not shown in that list. Such elements may appear in a different view of the list.

Group properties

A property string may be shown as a element of the list, in which case it applies to all of the succeeding list elements until another property string appears as a list element. This is equivalent to attaching the property string to each of the list elements individually. The property string does not designate a model element. Examples of this usage include indicating a stereotype and specifying visibility. Keyword strings may also be used in a similar way to qualify subsequent list elements.

Compartment name

A compartment may display a name to indicate which kind of compartment it is. The name is displayed in a distinctive font centered at the top of the compartment. This capability is useful if some compartments are omitted or if additional user-defined compartments are added. For a Class, the predefined compartments are named **attributes** and **operations**. An example of a user-defined compartment might be **requirements.** The name compartment in a class must always be present; therefore, it does not require or permit a compartment name.

3.22.2 Presentation Options

A tool may present the list elements in a sorted order, in which case the inherent ordering of the elements is not visible. A sort is based on some internal property and does not indicate additional model information. Example sort rules include:

- alphabetical order,
- ordering by stereotype (such as constructors, destructors, then ordinary methods),
- ordering by visibility (public, then protected, then private), etc.

The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be drawn about the presence or absence of elements that do not meet the criterion. However, the ellipsis notation is available to show that invisible elements exist. It is a tool responsibility whether and how to indicate the presence of either local or global filtering, although a stand-alone diagram should have some indication of such filtering if it is to be understandable.

If a compartment is suppressed, no inference can be drawn about the presence or absence of its elements. An empty compartment indicates that no elements meet the selection filter (if any).

Note that attributes may also be shown by composition (see Figure 3-25 on page 3-67).

3.22.3 Example

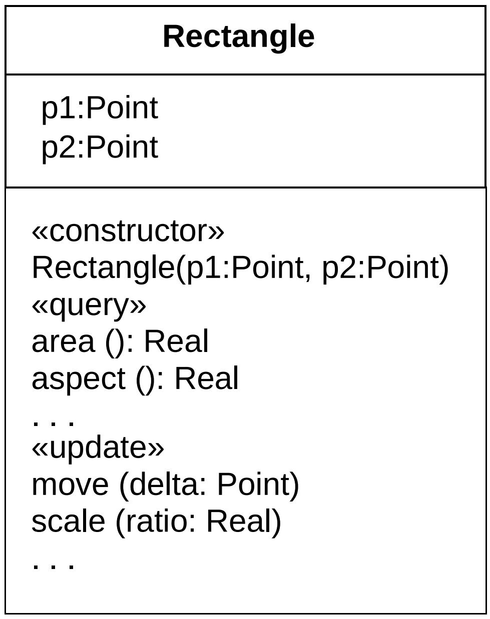


Figure 3-9 Stereotype Keyword Applied to Groups of List Elements

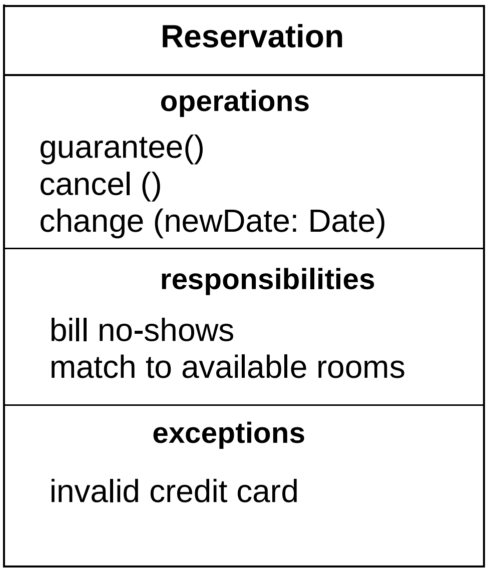


Figure 3-10 Compartments with Names

3.22.4 Mapping

The entries in a list compartment map into a list of ModelElements, one for each list entry. The ordering of the ModelElements matches the list compartment entries (unless the list compartment is sorted in some way). In this case, no implication about the ordering of the Elements can be made (the ordering can be seen by turning off sorting). However, a list entry string that is a stereotype indication (within guillemets) or a property indication (within braces) does not map into a separate ModelElement. Instead, the corresponding property applies to each subsequent ModelElement until the appearance of a different stand-alone stereotype or property indicator. The property specifications are conceptually duplicated for each list Element, although a tool might maintain an internal mechanism to store or modify them together. The presence of an ellipsis ("...") as a list entry implies that the semantic model contains at least one Element with corresponding properties that is not visible in the list compartment.

3.23 Attribute

Used to show attributes in classes. A similar syntax is used to specify qualifiers, template parameters, operation parameters, and so on (some of these omit certain terms).

3.23.1 Semantics

Note that an attribute is semantically equivalent to a composition association; however, the intent and usage is normally different.

The type of an attribute is a TypeExpression. It may resolve to a class name or it may be complex, such as array[String] of Point. In any case, the details of the attribute type expressions are not specified by UML. They depend on the expression syntax supported by the particular specification or programming language being used.

3.23.2 Notation

An attribute is shown as a text string that can be parsed into the various properties of an attribute model element. The default syntax is:

visibility name : *type-expression* = *initial-value* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  \# protected visibility

  \- private visibility

The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). A tool should assign visibilities to new attributes even if the visibility is not shown. The visibility marker is a shorthand for a full *visibility* property specification string.

Visibility may also be specified by keywords (*public, protected, private*). This form is used particularly when it is used as an inline list element that applies to an entire block of attributes.

Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string that represents the name of the attribute.
- Where *type-expression* is a language-dependent specification of the implementation type of an attribute.
- Where *initial-value* is a language-dependent expression for the initial value of a newly created object. The initial value is optional (the equal sign is also omitted). An explicit constructor for a new object may augment or modify the default initial value.
- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope attribute is shown by underlining the name and type expression string; otherwise, the attribute is instance-scope. The notation justification is that a class-scope attribute is an instance value in the executing system, just as an object is an instance value, so both may be designated by underlining. An instance-scope attribute is not underlined; that is the default.

<u>*class-scope-attribute*</u>

There is no symbol for whether an attribute is changeable (the default is changeable). A nonchangeable attribute is specified with the property “{frozen}”.

In the absence of a multiplicity indicator, an attribute holds exactly 1 value. Multiplicity may be indicated by placing a multiplicity indicator in brackets after the attribute name, for example:

colors [3]: Color
points [2..*]: Point

Note that a multiplicity of 0..1 provides for the possibility of null values: the absence of a value, as opposed to a particular value from the range. For example, the following declaration permits a distinction between the *null* value and the empty string:

name [0..1]: String

A stereotype keyword in guillemets precedes the entire attribute string, including any visibility indicators. A property list in braces follows the entire attribute string.

3.23.3 Presentation Options

The type expression may be suppressed (but it has a value in the model).

The initial value may be suppressed, and it may be absent from the model. It is a tool responsibility whether and how to show this distinction.

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

A tool may show the individual fields of an attribute as columns rather than a continuous string.

The syntax of the attribute string can be that of a particular programming language, such as C++ or Smalltalk. Specific tagged properties may be included in the string.

Particular attributes within a list may be suppressed (see “List Compartment” on page 3-29).

3.23.4 Style Guidelines

Attribute names typically begin with a lowercase letter. Attribute names are in plain face.

3.23.5 Example

+size: Area = (100,100)
#visibility: Boolean = invisible
<u>+default-size: Rectangle</u>
<u>#maximum-size: Rectangle</u>
-xptr: XWindowPtr

3.23.6 Mapping

A string entry within the attribute compartment maps into an Attribute within the Class representing the class symbol. The properties of the attribute map in accord with the preceding descriptions. If the visibility is absent, then no conclusion can be drawn about the Attribute visibilities unless a filter is in effect (e.g., only public attributes shown). Likewise, if the type or initial value are omitted. The omission of an underline always indicates an instance-scope attribute. The omission of multiplicity denotes a multiplicity of 1.

Any properties specified in braces following the attribute string map into properties on the Attribute. In addition, any properties specified on a previous stand-alone property specification entry apply to the current Attribute (and to others).

3.24 Operation

Used to show operations defined on classes. Also used to show methods supplied by classes.

3.24.1 Operation

An operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments.

3.24.2 Notation

An operation is shown as a text string that can be parsed into the various properties of an operation model element. The default syntax is:

visibility name (*parameter-list*) : *return-type-expression* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  \# protected visibility

  \- private visibility

The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). The visibility marker is a shorthand for a full *visibility* property specification string.

Visibility may also be specified by keywords (*public*, *protected*, *private*). This form is used particularly when it is used as an inline list element that applies to an entire block of operations.

Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string.
- Where *return-type-expression* is a language-dependent specification of the implementation type or types of the value returned by the operation. The return-type is omitted if the operation does not return a value (C++ void). A list of expressions may be supplied to indicate multiple return values.
- Where *parameter-list* is a comma-separated list of formal parameters, each specified using the syntax:

  kind name : *type-expression* = *default-value*

  - where *kind* is **in, out,** or **inout**, with the default **in** if absent.
  - where *name* is the name of a formal parameter.
  - where *type-expression* is the (language-dependent) specification of an implementation type.
  - where *default-value* is an optional value expression for the parameter, expressed in and subject to the limitations of the eventual target language.
- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope operation is shown by underlining the name and type expression string. An instance-scope operation is the default and is not marked.

An operation that does not modify the system state (one that has no side effects) is specified by the property "{query}"; otherwise, the operation may alter the system state, although there is no guarantee that it will do so.

The concurrency semantics of an operation are specified by a property string with one of the names: *sequential, guarded, concurrent*. In the absence of a specification, the concurrency semantics are undefined and must be assumed to be sequential in the worst case.

The top-most appearance of an operation signature declares the operation on the class (and inherited by all of its descendents). If this class does not implement the operation (i.e., does not supply a method), then the operation may be marked as "{abstract}" or the operation signature may be italicized to indicate that it is abstract. Any subordinate appearances of the operation signature indicate that the subordinate class implements a method on the operation. (The specification of "{abstract}" or italics on a subordinate class would not indicate a method, but this usage of the notation would be poor form.)

The actual text or algorithm of a method may be indicated in a note attached to the operation entry.

An operation entry with the stereotype «signal» indicates that the class accepts the given signal. The syntax is identical to that of an operation.

The specification of operation behavior is given as a note attached to the operation. The text of the specification should be enclosed in braces if it is a formal specification in some language (a semantic Constraint); otherwise, it should be plain text if it is just a natural-language description of the behavior (a Comment).

A stereotype keyword in guillemets precedes the entire operation string, including any visibility indicators. A property list in braces follows the entire operation string.

3.24.3 Presentation Options

The argument list and return type may be suppressed (together, not separately).

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

The syntax of the operation signature string can be that of a particular programming language, such as C++ or Smalltalk. Specific tagged properties may be included in the string.

3.24.4 Style Guidelines

Operation names typically begin with a lowercase letter. Operation names are in plain face. An abstract operation may be shown in italics.

3.24.5 Example

+display (): Location
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

Figure 3-11 Operation List with a Variety of Operations

3.24.6 Mapping

A string entry within the operation compartment maps into an Operation or a Method within the Class representing the class symbol. The properties of the operation map in accordance with the preceding descriptions. See the description of "Attribute" on page 3-32 for additional details.

The topmost appearance of an operation specification in a class hierarchy maps into an Operation definition in the corresponding Class or Interface. Interfaces do not have methods. In a Class, each appearance of an operation entry maps into the presence of a Method in the corresponding Class, unless the operation entry contains the {abstract} property (including use of conventions such as italics for abstract operations). If an abstract operation entry appears within a hierarchy in which the same operation has already been defined in an ancestor, it has no effect but is not an error unless the declarations are inconsistent.

Note that the operation string entry does not specify the body of a method.

3.24.7 Signal Reception

If the objects of a class accept and respond to a given signal, that fact can be indicated using the same syntax as an operation with the keyword «signal». The response of the object to the reception of the signal is shown with a state machine. Among other uses, this notation can show the response of objects of a class to error conditions and exceptions, which should be modeled as signals.

3.25 Type Vs. Implementation Class

3.25.1 Semantics

Classes can be specialized by stereotypes into Types and Implementation Classes (although they can be left undifferentiated as well). A Type characterizes a changeable role that an object may adopt and later abandon. An Implementation Class defines the physical data structure and procedures of an object as implemented in traditional languages (C++, Smalltalk, etc.). An object may have multiple Types (which may change dynamically) but only one ImplementationClass (which is fixed). Although the usage of Types and ImplementationClasses is different, their internal structure is the same, so they are modeled as stereotypes of Class. All kinds of Class require that a subclass fully support the features of the superclass, including support for all inherited attributes, associations, and operations.

3.25.2 Notation

An undifferentiated class is shown with no stereotype. A type is shown with the stereotype "«type»". An implementation class is shown with the stereotype "«implementation class»". A tool is also free to allow a default setting for an entire diagram, in which case all of the class symbols without explicit stereotype indications map into Classes with the default stereotype. This might be useful for a model that is close to the programming level.

The implementation of a type by an implementation class is modeled as the Realizes relationship, shown as a dashed line with a solid triangular arrowhead (a dashed "generalization arrow"). This symbol implies inheritance of operations, but not of structure (attributes or associations).

3.25.3 Example

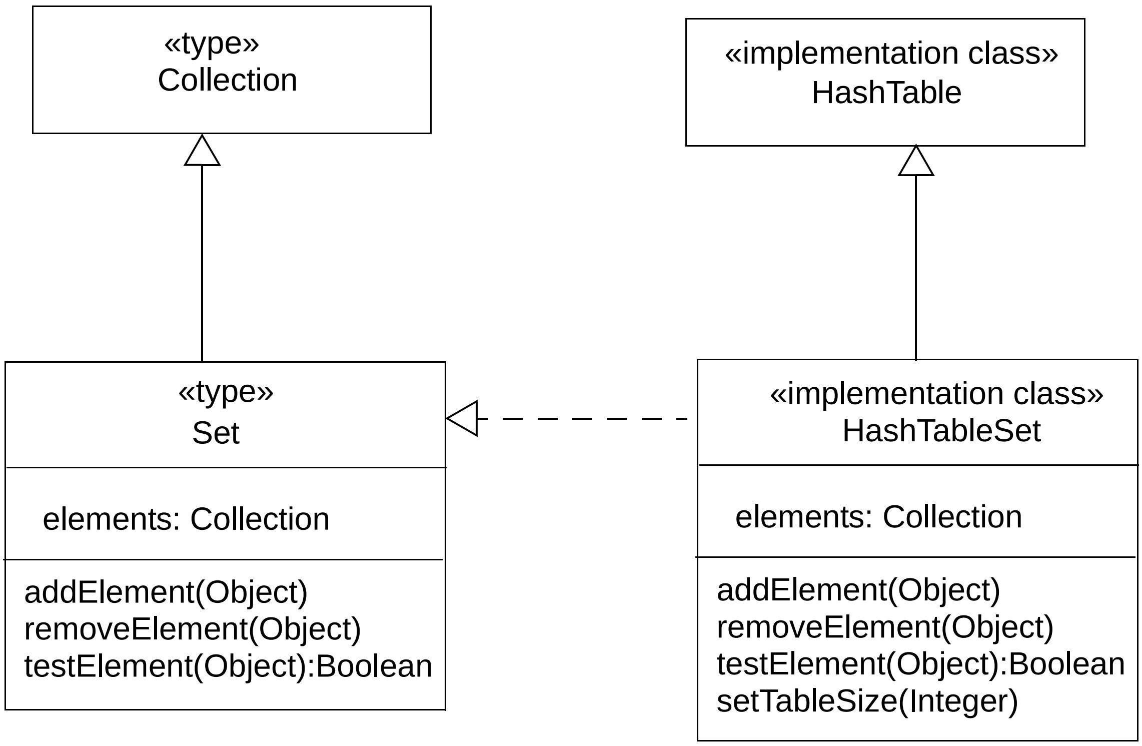


Figure 3-12 Notation for Types and Implementation Classes

3.25.4 Mapping

A class symbol with a stereotype (including "type" and "implementation class") maps into a Class with the corresponding stereotype. A class symbol without a stereotype maps into a Class with the default stereotype for the diagram (if a default has been defined by the modeler or tool); otherwise, it maps into a Class with no stereotype. This symbol is normally used between a class and an interface, but may also be used between any two classifiers to show inheritance of operations only without inheritance of attributes or associations.

3.26 Interfaces

3.26.1 Semantics

An interface is a specifier for the externally-visible operations of a class, component, or other entity (including summarization units such as packages) without specification of internal structure. Each interface often specifies only a limited part of the behavior of an actual class. Interfaces do not have implementation. They lack attributes, states, or associations, they only have operations. Interfaces may have generalization relationships. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel (both are Classifiers).

3.26.2 Notation

An interface is a Classifer and may also be shown using the full rectangle symbol with compartments and the keyword «interface». A list of operations supported by the interface is placed in the operation compartment. The attribute compartment may be omitted because it is always empty.

An interface may also be displayed as a small circle with the name of the interface placed below the symbol. The circle may be attached by a solid line to classes that support it (also to higher-level containers, such as packages that contain the classes). This indicates that the class provides all of the operations in the interface type (and possibly more). The operations provided are not shown on the circle notation; use the full rectangle symbol to show the list of operations. A class that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. The dashed arrow implies that the class requires no more than the operations specified in the interface; the client class is not required to actually use *all* of the interface operations.

The Realizes relationship from a class to an interface that it supports is shown by a dashed line with a solid triangular arrowhead (a “dashed generalization symbol”). This is the same notation used to indicate realization of a type by an implementation class. In fact, this symbol can be used between any two classifier symbols, with the meaning that the client (the one at the tail of the arrow) supports at least all of the operations defined in the supplier (the one at the head of the arrow), but with no necessity to support any of the data structure of the supplier (attributes and associations).

3.26.3 Example

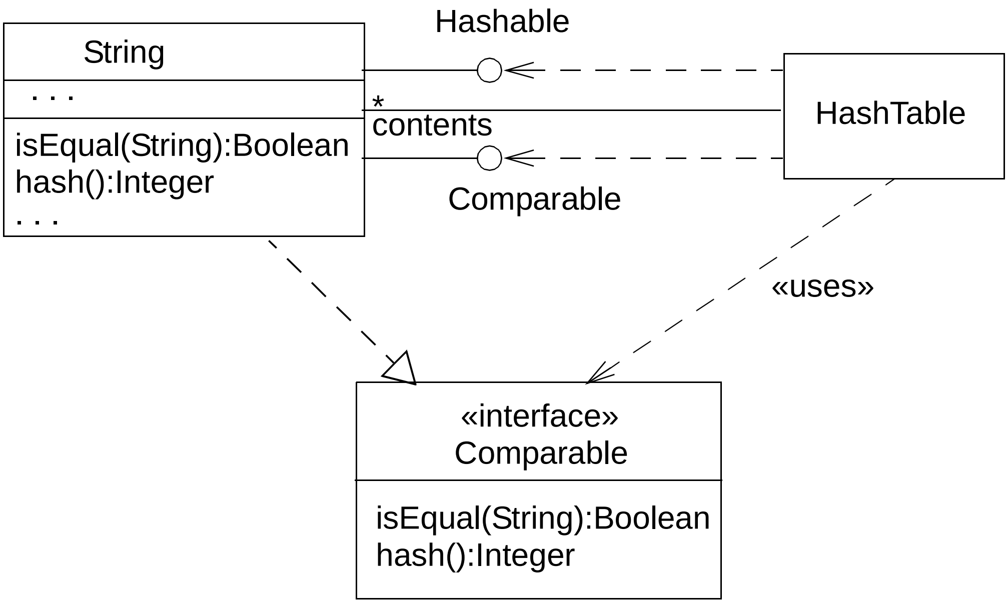


Figure 3-13 Interface Notation on Class Diagram

3.26.4 Mapping

A class rectangle symbol with stereotype «interface», or a circle on a class diagram, maps into an Interface element with the name given by the symbol. The operation list of a rectangle symbol maps into the list of Operation elements of the Interface.

A dashed generalization arrow from a class symbol to an interface symbol, or a solid line connecting a class symbol and an interface circle, maps into a realization-specification relationship between the corresponding Class and Interface elements. A dependency arrow from a class symbol to an interface symbol maps into a «uses» dependency between the corresponding Class and Interface.

3.27 Parameterized Class (Template)

3.27.1 Semantics

A template is the descriptor for a class with one or more unbound formal parameters. It defines a family of classes, each class specified by binding the parameters to actual values. Typically, the parameters represent attribute types; however, they can also represent integers, other types, or even operations. Attributes and operations within the template are defined in terms of the formal parameters so they too become bound when the template itself is bound to actual values.

A template is not a directly-usable class because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a class. Only a class can be s superclass or the target of an association (a one-way association *from* the template *to* another class is permissible, however). A template may be a subclass of an ordinary class. This implies that all classes formed by binding it are subclasses of the given superclass.

Parameterization can be applied to other ModelElements, such as Collaborations or even entire Packages. The description given here for classes applies to other kinds of modeling elements in the obvious way.

3.27.2 Notation

A small dashed rectangle is superimposed on the upper right-hand corner of the rectangle for the class (or to the symbol for another modeling element). The dashed rectangle contains a parameter list of formal parameters for the class and their implementation types. The list must not be empty, although it might be suppressed in the presentation. The name, attributes, and operations of the parameterized class appear as normal in the class rectangle; however, they may also include occurrences of the formal parameters. Occurrences of the formal parameters can also occur inside of a context for the class, for example, to show a related class identified by one of the parameters.

3.27.3 Presentation Options

The parameter list may be comma-separated or it may be one per line.

Parameters are restricted attributes, shown as strings with the syntax

name : *type*

- Where *name* is an identifier for the parameter with scope inside the template.
- Where *type* is a string designating a *TypeExpression* for the parameter.

If the type name is omitted, it is assumed to be a type expression that resolves to a classifier, such as a class name or a data type. Other parameter types (such as Integer) must be explicitly shown, they must resolve to valid type expressions.

3.27.4 Example

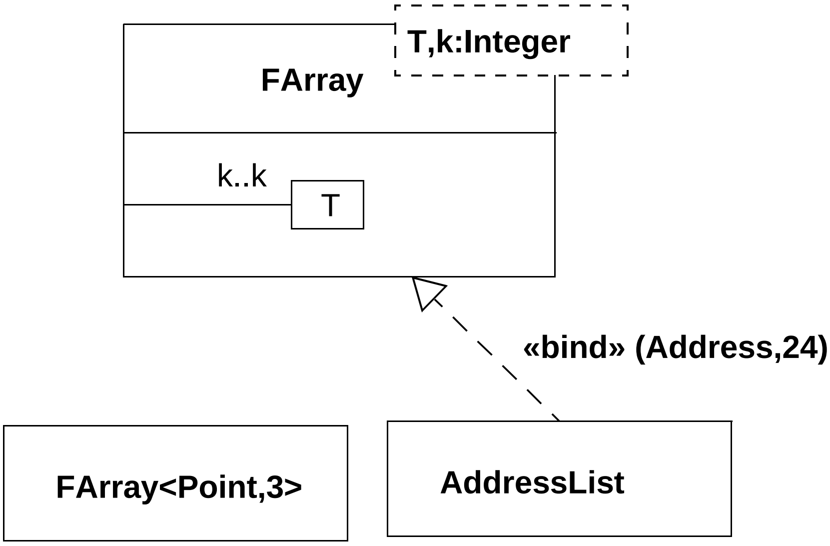


Figure 3-14 Template Notation with Use of Parameter as a Reference

3.27.5 Mapping

The addition of the template dashed box to a symbol causes the addition of the parameter names in the list as ModelElements within the Namespace of the ModelElement corresponding to the base symbol. Each of the parameter ModelElements has the templateParameter association to the Namespace.

3.28 Bound Element

3.28.1 Semantics

A template cannot be used directly in an ordinary relationship such as generalization or association, because it has a free parameter that is not meaningful outside of a scope that declares the parameter. To be used, a template's parameters must be *bound* to actual values. The actual value for each parameter is an expression defined within the scope of use. If the referencing scope is itself a template, then the parameters of the referencing template can be used as actual values in binding the referenced template. The parameter names in the two templates cannot be assumed to correspond because they have no scope outside of their respective templates.

3.28.2 Notation

A bound element is indicated by a text syntax in the name string of an element, as follows:

Template-name ‘<‘ *value-list* ‘>’

- Where *value-list* is a comma-delimited non-empty list of value expressions.
- Where *Template-name* is identical to the name of a template.

For example, VArray<Point,3> designates a class described by the template Varray.

The number and type of values must match the number and type of the template parameters for the template of the given name.

The bound element name may be used anywhere that an element name of the parameterized kind could be used. For example, a bound class name could be used within a class symbol on a class diagram, as an attribute type, or as part of an operation signature.

Note that a bound element is fully specified by its template; therefore, its content may not be extended. Declaration of new attributes or operations for classes is not permitted, for example, but a bound class could be subclassed and the subclass extended in the usual way.

The relationship between the bound element and its template alternatively may be shown by a Dependency relationship with the keyword «bind». The arguments are shown in parentheses after the keyword. In this case, the bound form may be given a name distinct from the template.

3.28.3 Style Guidelines

The attribute and operation compartments are normally suppressed within a bound class, because they must not be modified in a bound template.

3.28.4 Example

See Figure 3-14 on page 3-43.

3.28.5 Mapping

The use of the bound element syntax for the name of a symbol maps into a Binding dependency between the dependent ModelElement (such as Class) corresponding to the bound element symbol and the provider ModelElement (again, such as Class) whose name matches the name part of the bound element without the arguments. If the name does not match a template element or if the number of arguments in the bound element does not match the number of parameters in the template, then the model is ill formed. Each argument in the bound element maps into a ModelElement bearing a templateArgument association to the Namespace of the bound element. The Binding relationship bears the list of actual argument values.

3.29 Utility

A utility is a grouping of global variables and procedures in the form of a class declaration. This is not a fundamental construct, but a programming convenience. The attributes and operations of the utility become global variables and procedures. A utility is modeled as a stereotype of a class.

3.29.1 Semantics

The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

3.29.2 Notation

Shown as the stereotype «utility» of Class. It may have both attributes and operations, all of which are treated as global attributes and operations.

3.29.3 Example

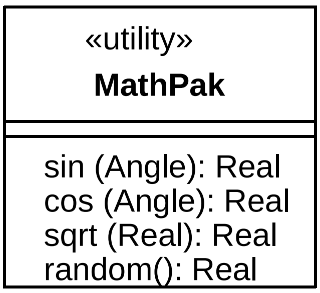


Figure 3-15 Notation for Utility

3.29.4 Mapping

This is not a special symbol. It simply maps into a Class element with the «utility» stereotype.

3.30 Metaclass

3.30.1 Semantics

A metaclass is a class whose instances are classes.

3.30.2 Notation

Shown as the stereotype «metaclass» of Class.

3.30.3 Mapping

This is not a special symbol. It simply maps into a Class element with the «metaclass» stereotype.

3.31 Class Pathnames

3.31.1 Notation

Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

package-name :: *class-name*

References to classes also appear in text expressions, most notably in type specifications for attributes and variables. In these places a reference to a class is indicated by simply including the name of the class itself, including a possible package name, subject to the syntax rules of the expression.

3.31.2 Example

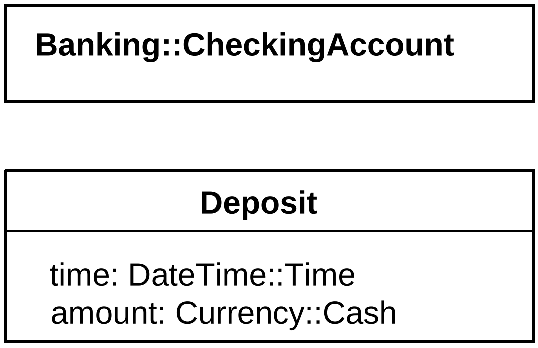


Figure 3-16 Pathnames for Classes in Other Packages

3.31.3 Mapping

A class symbol whose name string is a pathname represents a reference to the Class with the given name inside the package with the given name. The name is assumed to be defined in the target package; otherwise, the model is ill formed. A Relationship from a symbol in the current package (i.e., the package containing the diagram and its mapped elements) to a symbol in another package is part of the current package.

3.32 Importing a Package

3.32.1 Semantics

A class in another package may be referenced. On the package level, the «import» dependency indicates that the contents of the target packages may be referenced by the client package or packages recursively embedded within it. The target references must have visibility sufficient for the referents. Visibilities may be specified on model elements and on packages. If a model element is nested inside one or more packages, the visibilities of the element and all of its containers combine according to the rule that the most restrictive visibility in the set is obtained. It is not possible to selectively export certain elements from within a nested package; the visibility of the outer package is applied to each element exported by an inner package. Imports are recursive within nested levels of packages. A descendent of a class requires at least "protected" visibility; any other class requires "public" visibility. (See the UML Semantics chapter for full details.)

Note that an import's dependency does not modify the namespace of the client or in any other way automatically create references; it merely grants permission to establish references. Note also that a tool could automatically create imports dependencies for users if desired when references are created.

3.32.2 Notation

The imports dependency is displayed as a dependency arrow from the referencing (client) package to the target (supplier) package containing the target of the references. The arrow has the stereotype «import». This dependency indicates that elements within the client package may legally reference elements within the supplier. The references must also satisfy visibility constraints specified by the supplier. Note that the dependency does not automatically create any references. It merely grants permission for them to be established.

3.32.3 Example

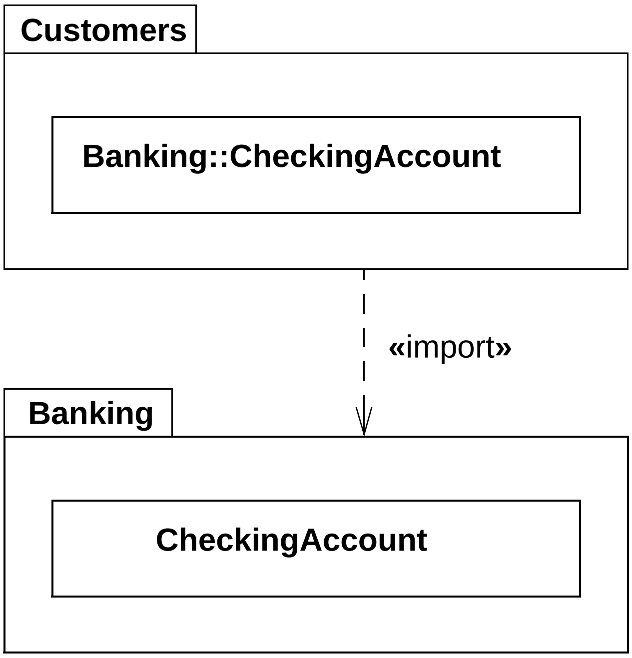


Figure 3-17 Imports Dependency Among Packages

3.32.4 Mapping

This is not a special symbol. It maps into a Dependency with the stereotype «import» between the two packages.

3.33 Object

3.33.1 Semantics

An object represents a particular instance of a class. It has identity and attribute values. The same notation also represents a role within a collaboration because roles have instance-like characteristics.

3.33.2 Notation

The object notation is derived from the class notation by underlining instance-level elements, as explained in the general comments in “Type-Instance Correspondence” on page 3-14.

An object shown as a rectangle with two compartments.

The top compartment shows the name of the object and its class, all underlined, using the syntax:

objectname : *classname*

The classname can include a full pathname of enclosing package, if necessary. The package names precede the classname and are separated by double colons. For example:

```
display_window: WindowingSystem::GraphicWindows::Window
```

A stereotype for the class may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. The stereotype for an object must match the stereotype for its class.

To show multiple classes that the object is an instance of, use a comma-separated list of classnames. These classnames must be legal for multiple classification (i.e., only one implementation class permitted, but multiple roles permitted).

To show the presence of an object in a particular state of a class, use the syntax:

objectname : *classname* '[' *statename-list* ']'

The list must be a comma-separated list of names of states that can legally occur concurrently.

The second compartment shows the attributes for the object and their values as a list. Each value line has the syntax:

attributename : *type* = *value*

The type is redundant with the attribute declaration in the class and may be omitted.

The value is specified as a literal value. UML does not specify the syntax for literal value expressions; however, it is expected that a tool will specify such a syntax using some programming language.

3.33.3 Presentation Options

The name of the object may be omitted. In this case, the colon should be kept with the class name. This represents an anonymous object of the given class given identity by its relationships.

The class of the object may be suppressed (together with the colon).

The attribute value compartment as a whole may be suppressed.

Attributes whose values are not of interest may be suppressed.

Attributes whose values change during a computation may show their values as a list of values held over time. This is a good opportunity for the use of animation by a tool (the values would change dynamically). An alternate notation is to show the same object more than once with a «becomes» relationship between them.

3.33.4 Style Guidelines

Objects may be shown on class diagrams. The elements on collaboration diagrams are not objects, because they describe many possible objects. They are instead roles that may be held by object. Objects in class diagrams serve mainly to show examples of data structures.

3.33.5 Variations

For a language such as *Self* in which operations can be attached to individual objects at run time, a third compartment containing operations would be appropriate as a language-specific extension.

3.33.6 Example

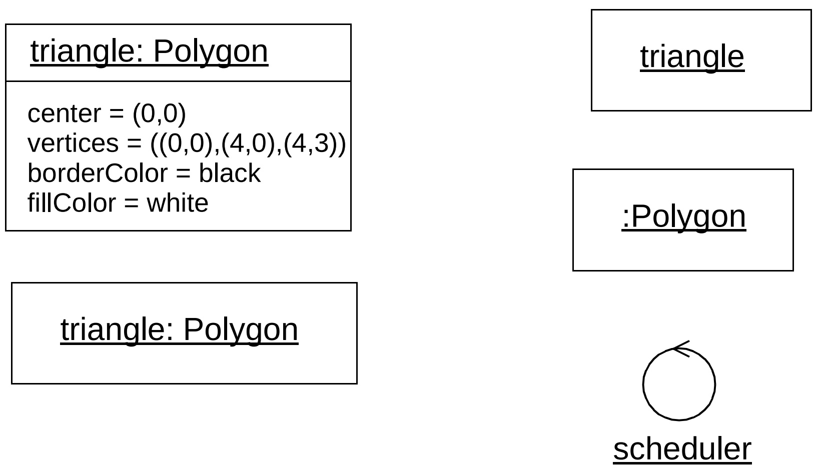


Figure 3-18 Objects

3.33.7 Mapping

The mapping of an object symbol depends on the diagram: Within a collaboration, it maps into a ClassifierRole of the corresponding Collaboration. The role has the name specified by the *objectname* portion of the symbol name string. The ClassifierRole has a type association to the Class whose name appears in the *classname* part of the symbol name string.

In an object diagram, or within an ordinary class diagram, it maps into an Object of the Class given by the *classname* part of the name string. The values of the attributes are given by the value expressions in the attribute list in the symbol.

3.34 Composite Object

3.34.1 Semantics

A composite object represents a high-level object made of tightly-bound parts. This is an instance of a composite class, which implies the composition aggregation between the class and its parts. A composite object is similar to (but simpler and more restricted than) a collaboration; however, it is defined completely by composition in a static model.

3.34.2 Notation

A composite object is shown as an object symbol. The name string of the composite object is placed in a compartment near the top of the rectangle (as with any object). The lower compartment holds the parts of the composite object instead of a list of attribute values. (However, even a list of attribute values may be regarded as the parts of a composite object, so there is not such a difference.) It is possible for some of the parts to be composite objects with further nesting.

3.34.3 Example

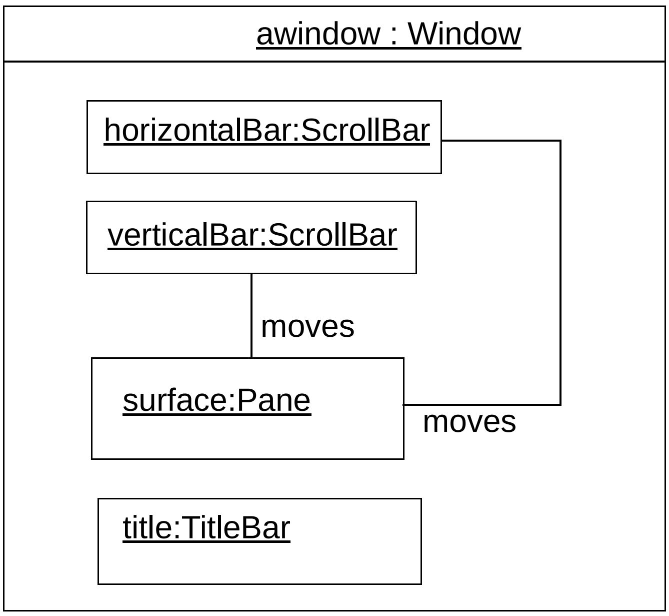


Figure 3-19 Composite Objects

3.34.4 Mapping

A composite object symbol maps into an Object of the given Class with composition links to each of the Objects and Links corresponding to the class box symbols, and association path symbols directly contained within the boundary of the composite object symbol (and not contained within another deeper boundary).

3.35 Association

Binary associations are shown as lines connecting two class symbols. The lines may have a variety of adornments to show their properties. Ternary and higher-order associations are shown as diamonds connected to class symbols by lines.

3.36 Binary Association

3.36.1 Semantics

A binary association is an association among exactly two classes (including the possibility of a reflexive association from a class to itself).

3.36.2 Notation

A binary association is drawn as a solid path connecting two class symbols (both ends may be connected to the same class, but the two ends are distinct). The path may consist of one or more connected segments. The individual segments have no semantic significance, but may be graphically meaningful to a tool in dragging or resizing an association symbol. A connected sequence of segments is called a *path*.

In a binary association, both ends may attach to the same class. The links of such an association may connect two different objects from the same class or one object to itself. The latter case is a *reflexive* association; it may be forbidden by a constraint if necessary.

The end of an association where it connects to a class is called an *association end*. Most of the interesting information about an association is attached to its roles.

The path may also have graphical adornments attached to the main part of the path itself. These adornments indicate properties of the entire association. They may be dragged along a segment or across segments, but must remain attached to the path. It is a tool responsibility to determine how close association adornments may approach a role so that confusion does not occur. The following kinds of adornments may be attached to a path.

association name

Designates the (optional) name of the association.

Shown as a name string near the path (but not near enough to an end to be confused with a rolename). The name string may have an optional small black solid triangle in it. The point of the triangle indicates the direction in which to read the name. The name-direction arrow has no semantics significance, it is purely descriptive. The classes in the association are ordered as indicated by the name-direction arrow.

Note – There is no need for a *name direction* property on the association model; the ordering of the classes within the association *is* the name direction. This convention works even with n-ary associations.

A stereotype keyword within guillemets may be placed above or in front of the association name. A property string may be placed after or below the association name.

association class symbol

Designates an association that has class-like properties, such as attributes, operations, and other associations. This is present if, and only if, the association is an association class. Shown as a class symbol attached to the association path by a dashed line.

The association path and the association class symbol represent the same underlying model element, which has a single name. The name may be placed on the path, in the class symbol, or on both (but they must be the same name).

Logically, the association class and the association are the same semantic entity; however, they are graphically distinct. The association class symbol can be dragged away from the line, but the dotted line must remain attached to both the path and the class symbol.

3.36.3 Presentation Options

When two paths cross, the crossing may optionally be shown with a small semicircular jog to indicate that the paths do not intersect (as in electrical circuit diagrams). Alternately, crossing can be unmarked but connections might be shown by small dots.

3.36.4 Style Guidelines

Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice.

3.36.5 Options

Or-association

An or-constraint indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string "{or}" labeling the dashed line. Any instance of the class may only participate in one of the associations at one time. Each rolename must be different. (This is simply a predefined use of the constraint notation.)

3.36.6 Example

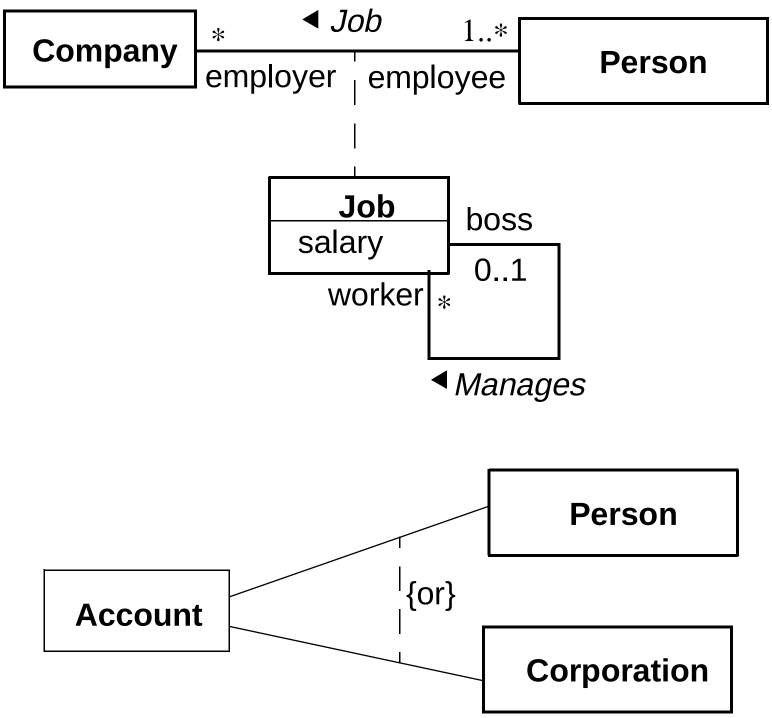


Figure 3-20 Association Notation

3.36.7 Mapping

An association path connecting two class symbols maps to an Association between the corresponding Classes. If there is an arrow on the association name, then the Class corresponding to the tail of the arrow is the first class and the Class corresponding to the head of the arrow is the second Class in the ordering of roles of the Association; otherwise, the ordering of roles in the association is undetermined. The adornments on the path map into properties of the Association as described above. The Association is owned by the package containing the diagram.

3.37 Association End

3.37.1 Semantics

An association end is simply an end of an association where it connects to a class. It is part of the association, not part of the class. Each association has two or more ends. Most of the interesting details about an association are attached to its ends. An association end is not a separable element, it is just a mechanical part of an association.

3.37.2 Notation

The path may have graphical adornments at each end where the path connects to the class symbol. These adornments indicate properties of the association related to the class. The adornments are part of the association symbol, not part of the class symbol. The end adornments are either attached to the end of the line, or near the end of the line, and must drag with it. The following kinds of adornments may be attached to an association end.

multiplicity

Specified by a text syntax. Multiplicity may be suppressed on a particular association or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself. In this case, it must be suppressed in the notation.

ordering

If the multiplicity is greater than one, then the set of related elements can be ordered or unordered. If no indication is given, then it is unordered (the elements form a set). Various kinds of ordering can be specified as a constraint on the association end. The declaration does not specify how the ordering is established or maintained. Operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:

- unordered - the elements form an unordered set. This is the default and need not be shown explicitly.
- ordered - the elements of the set are ordered into a list. It is still a set and duplicates are prohibited. This generic specification includes all kinds of ordering. This may be specified by the keyword syntax “{ordered}”.

An ordered relationship may be implemented in various ways; however, this is normally specified as a language-specified code generation property to select a particular implementation. An implementation extension might substitute the data structure to hold the elements for the generic specification “ordered.”

At implementation level, sorting may also be specified. It does not add new semantic information, but it expresses a design decision:

- sorted - the elements are sorted based on their internal values. The actual sorting rule is best specified as a separate constraint.

qualifier

Qualifier is optional, but not suppressible.

navigability

An arrow may be attached to the end of the path to indicate that navigation is supported toward the class attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. To be totally explicit, arrows may be shown whenever navigation is supported in a given direction. In practice, it is often convenient to suppress some of the arrows and just show exceptional situations. See “Presentation Options” on page 3-27 for details.

aggregation indicator

A hollow diamond is attached to the end of the path to indicate aggregation. The diamond may not be attached to both ends of a line, but it need not be present at all. The diamond is attached to the class that is the aggregate. The aggregation is optional, but not suppressible.

If the diamond is filled, then it signifies the strong form of aggregation known as *composition.*

rolename

A name string near the end of the path. It indicates the role played by the class attached to the end of the path near the rolename. The rolename is optional, but not suppressible.

interface specifier

The name of a Classifier with the syntax:

‘:’ *classifiername*

It indicates the behavior expected of an associated object by the related object. In other words, the interface specifier specifies the behavior required to enable the association. In this case, the actual class usually provides more functionality than required for the particular association (since it may have other responsibilities).

The use of a rolename and interface specifier are equivalent to creating a small collaboration that includes just an association and two roles, whose structure is defined by the rolename and role classifier on the original association. Therefore, the original association and classes are a use of the collaboration. The original class must be compatible with the interface specifier (which can be an interface or a type).

If an interface specifier is omitted, then the association may be used to obtain full access to the associated class.

changeability

If the links are changeable (can be added, deleted, and moved), then no indicator is needed. The property {frozen} indicates that no links may be added, deleted, or moved from an object (toward the end with the adornment) after the object is created and initialized. The property {addOnly} indicates that additional links may be added (presumably, the multiplicity is variable); however, links may not be modified or deleted.

visibility

Specified by a visibility indicator (‘+’, ‘#’, ‘-’ or explicit keyword such as {public}) in front of the rolename. Specifies the visibility of the association traversing in the direction toward the given rolename. See “Attribute” on page 3-32 for details of visibility specification.

Other properties can be specified for association roles, but there is no graphical syntax for them. To specify such properties, use the constraint syntax near the end of the association path (a text string in braces). Examples of other properties include mutability.

3.37.3 Presentation Options

If there are two or more aggregations to the same aggregate, they may be drawn as a tree by merging the aggregation end into a single segment. This requires that all of the adornments on the aggregation ends be consistent. This is purely a presentation option, there are no additional semantics to it.

Various options are possible for showing the navigation arrows on a diagram. These can vary from time to time by user request or from diagram to diagram.

- Presentation option 1: Show all arrows. The absence of an arrow indicates navigation is not supported.
- Presentation option 2: Suppress all arrows. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.
- Presentation option 3: Suppress arrows for associations with navigability in both directions, show arrows only for associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from no-way navigation; however, the latter case is normally rare or nonexistent in practice. This is yet another example of a situation in which some information is suppressed from a view.

3.37.4 Style Guidelines

If there are multiple adornments on a single role, they are presented in the following order, reading from the end of the path attached to the class toward the bulk of the path:

- qualifier
- aggregation symbol
- navigation arrow

Rolenames and multiplicity should be placed near the end of the path so that they are not confused with a different association. They may be placed on either side of the line. It is tempting to specify that they will always be placed on a given side of the line (clockwise or counterclockwise), but this is sometimes overridden by the need for clarity in a crowded layout. A rolename and a multiplicity may be placed on opposite sides of the same role, or they may be placed together (for example, “* employee”).

3.37.5 Example

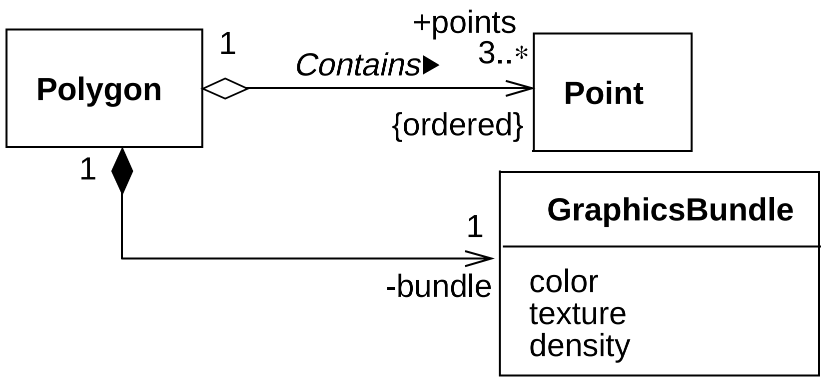


Figure 3-21 Various Adornments on Association Roles

3.37.6 Mapping

The adornments on the end of an association path map into properties of the corresponding role of the Association. In general, implications cannot be drawn from the absence of an adornment (it may simply be suppressed) but see the preceding descriptions for details.

3.38 Multiplicity

3.38.1 Semantics

A multiplicity item specifies 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 specification is a subset of the open set of nonnegative integers.

3.38.2 Notation

A multiplicity specification is shown as a text string comprising a comma-separated sequence of integer intervals, where an interval represents a (possibly infinite) range of integers, in the format:

lower-bound .. upper-bound

where *lower-bound* and *upper-bound* are literal integer values, specifying the closed (inclusive) range of integers from the lower bound to the upper bound. In addition, the star character (*) may be used for the upper bound, denoting an unlimited upper bound. In a parameterized context (such as a template), the bounds could be expressions but they must evaluate to literal integer values for any actual use. Unbound expressions that do not evaluate to literal integer values are not permitted.

If a single integer value is specified, then the integer range contains the single integer value.

If the multiplicity specification comprises a single star (*), then it denotes the unlimited nonnegative integer range, that is, it is equivalent to *..* = 0..* (zero or more).

A multiplicity of 0..0 is meaningless as it would indicate that no instances can occur.

Expressions in some specification language can be used for multiplicities, but they must resolve to fixed integer ranges within the model (i.e., no dynamic evaluation of expressions, essentially the same rule on literal values as most programming languages).

3.38.3 Style Guidelines

Preferably, intervals should be monotonically increasing. For example, “1..3,7,10” is preferable to “7,10,1..3”.

Two contiguous intervals should be combined into a single interval. For example, “0..1” is preferable to “0,1”.

3.38.4 Example

0..1

1

0..*

*

1..*

1..6

1..3,7..10,15,19..*

3.38.5 Mapping

A multiplicity string maps into a Multiplicity value. Duplications or other nonstandard presentation of the string itself have no effect on the mapping. Note that Multiplicity is a value and not an object. It cannot stand on its own, but is the value of some element property.

3.39 Qualifier

3.39.1 Semantics

A qualifier is an attribute or list of attributes whose values serve to partition the set of objects associated with an object across an association. The qualifiers are attributes of the association.

3.39.2 Notation

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the class that it connects to. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle drags with the path segments. The qualifier is attached to the source end of the association. An object of the source class, together with a value of the qualifier, uniquely select a partition in the set of target class objects on the other end of the association (i.e., every target falls into exactly one partition).

The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value. Common values include:

- “0..1” (a unique value may be selected, but every possible qualifier value does not necessarily select a value).
- “1” (every possible qualifier value selects a unique target object; therefore, the domain of qualifier values must be finite).

- “*” (the qualifier value is an index that partitions the target objects into subsets).

The qualifier attributes are drawn within the qualifier box. There may be one or more attributes shown one to a line. Qualifier attributes have the same notation as class attributes, except that initial value expressions are not meaningful.

It is permissible (although somewhat rare), to have a qualifier on each end of a single association.

3.39.3 Presentation Options

A qualifier may not be suppressed (it provides essential detail whose omission would modify the inherent character of the relationship).

A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

3.39.4 Style Guidelines

The qualifier rectangle should be smaller than the attached class rectangle, although this is not always practical.

3.39.5 Example

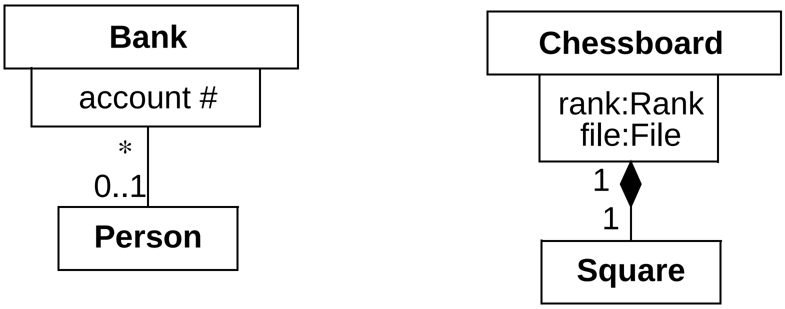


Figure 3-22 Qualified Associations

3.39.6 Mapping

The presence of a qualifier box on an end of an association path maps into a Qualifier on the corresponding Association Role. Each attribute entry string inside the qualifier box maps into an Attribute of the Qualifier.

3.40 Association Class

3.40.1 Semantics

An association class is an association that also has class properties (or a class that has association properties). Even though it is drawn as an association and a class, it is really just a single model element.

3.40.2 Notation

An association class is shown as a class symbol (rectangle) attached by a dashed line to an association path. The name in the class symbol and the name string attached to the association path are redundant and should be the same. The association path may have the usual adornments on either end. The class symbol may have the usual contents. There are no adornments on the dashed line.

3.40.3 Presentation Options

The class symbol may be suppressed. It provides subordinate detail whose omission does not change the overall relationship. The association path may not be suppressed.

3.40.4 Style Guidelines

The attachment point should not be near enough to either end of the path that it appears to be attached to, the end of the path, or to any of the role adornments.

Note that the association path and the association class are a single model element and have a single name. The name can be shown on the path, the class symbol, or both. If an association class has only attributes, but no operations or other associations, then the name may be displayed on the association path and omitted from the association class symbol to emphasize its “association nature.” If it has operations and other associations, then the name may be omitted from the path and placed in the class rectangle to emphasize its “class nature.” In neither case are the actual semantics different.

3.40.5 Example

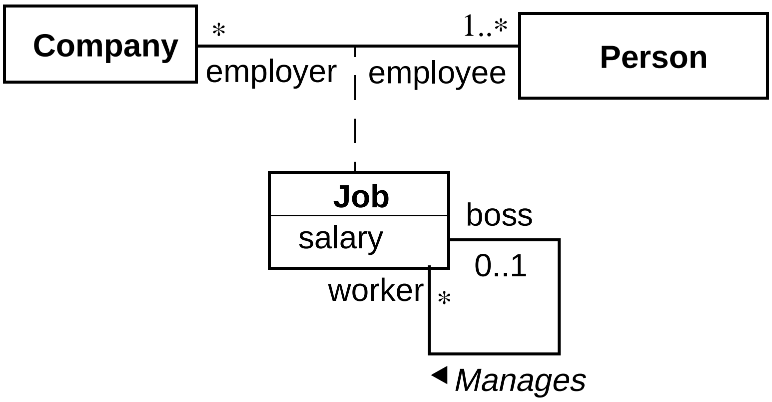


Figure 3-23 Association Class

3.40.6 Mapping

An association path connecting two class boxes connected by a dashed line to another class box maps into a single Association Class element. The name of the Association Class element is taken from the association path, the attached class box, or both (they must be consistent if both are present). The Association properties map from the association path, as specified previously. The Class properties map from the class box, as specified previously. Any constraints or properties placed on either the association path or attached class box apply to the Association Class itself, they must not conflict.

3.41 N-ary Association

3.41.1 Semantics

An n-ary association is an association among three or more classes (a single class may appear more than once). Each instance of the association is an n-tuple of values from the respective classes. A binary association is a special case with its own notation.

Multiplicity for n-ary associations may be specified, but is less obvious than binary multiplicity. The multiplicity on a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.

An n-ary association may not contain the aggregation marker on any role.

3.41.2 Notation

An n-ary association is shown as a large diamond (that is, large compared to a terminator on a path) with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. Role adornments may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted.

An association class symbol may be attached to the diamond by a dashed line. This indicates an n-ary association that has attributes, operations, and/or associations.

3.41.3 Style Guidelines

Usually the lines are drawn from the points on the diamond or the midpoint of a side.

3.41.4 Example

This example shows the record of a team in each season with a particular goalkeeper. It is assumed that the goalkeeper might be traded during the season and can appear with different teams.

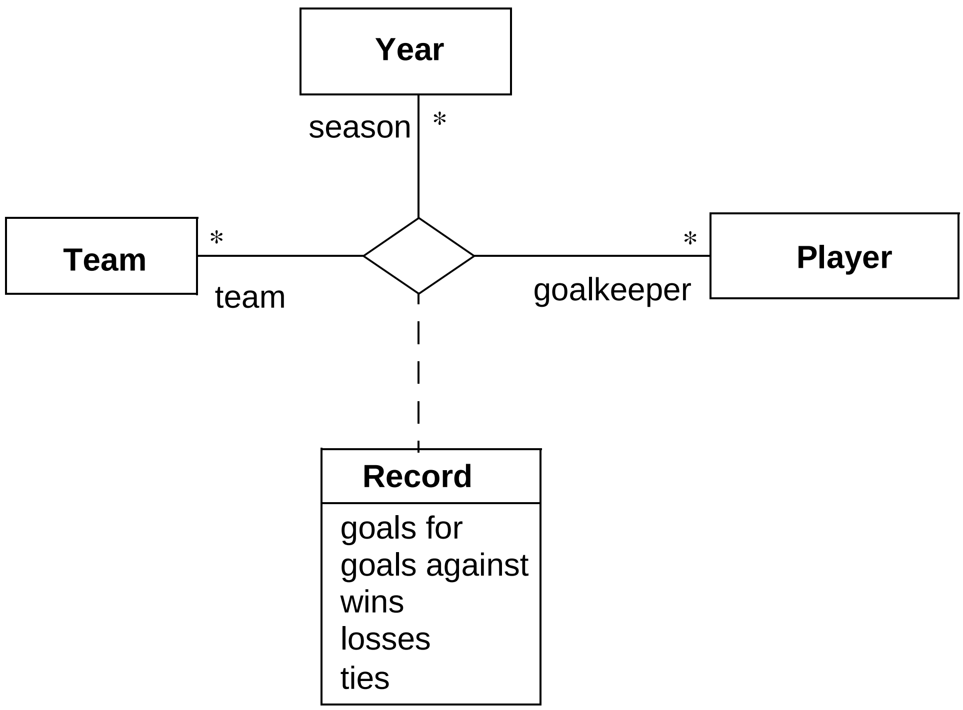


Figure 3-24 Ternary association that is also an association class

3.41.5 Mapping

A diamond attached to some number of class boxes by solid lines maps into an N-ary Association whose roles are corresponding Classes. The ordering of the Classes in the Association is indeterminate from the diagram. If a class box is attached to the diamond by a dashed line, then the corresponding Class supplies the class properties for an N-ary Association Class.

3.42 Composition

3.42.1 Semantics

Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). See the Semantics chapters (2-5) for further details.

The parts of a composition may include classes and associations. The meaning of an association in a composition is that any tuple of objects connected by a single link must all belong to the *same* container object.

3.42.2 Notation

Composition may be shown by a solid filled diamond as an association role adornment. Alternately, UML provides a graphically-nested form that is more convenient for showing composition in many cases.

Instead of using binary association paths using the composition aggregation adornment, composition may be shown by graphical nesting of the symbols of the elements for the parts within the symbol of the element for the whole. A nested class-like element may have a multiplicity within its composite element. The multiplicity is shown in the upper right corner of the symbol for the part. If the multiplicity mark is omitted, then the default multiplicity is many. This represents its multiplicity as a part within the composite class. A nested element may have a rolename within the composition; the name is shown in front of its type in the syntax:

rolename ‘:’ *classname*

This represents its rolename within its composition association to the composite.

Alternately, composition is shown by a solid-filled diamond adornment on the end of an association path attached to the element for the whole. The multiplicity may be shown in the normal way.

Note that attributes are, in effect, composition relationships between a class and the classes of its attributes.

An association drawn entirely within a border of the composite is considered to be part of the composition. Any objects on a single link of it must be from the same composite. An association drawn such that its path breaks the border of the composite is not considered to be part of the composition. Any objects on a single link of it may be from the same or different composites.

Note that the notation for composition resembles the notation for collaboration. A composition may be thought of as a collaboration in which all of the participants are parts of a single composite object.

3.42.3 Design Guidelines

This notation is applicable to "class-like" model elements (e.g., classes, types, nodes, processes, etc.).

Note that a class symbol is a composition of its attributes and operations. The class symbol may be thought of as an example of the composition nesting notation (with some special layout properties). However, attribute notation subordinates the attributes strongly within the class; therefore, it should be used when the structure and identity of the attribute objects themselves is unimportant outside the class.

3.42.4 Example

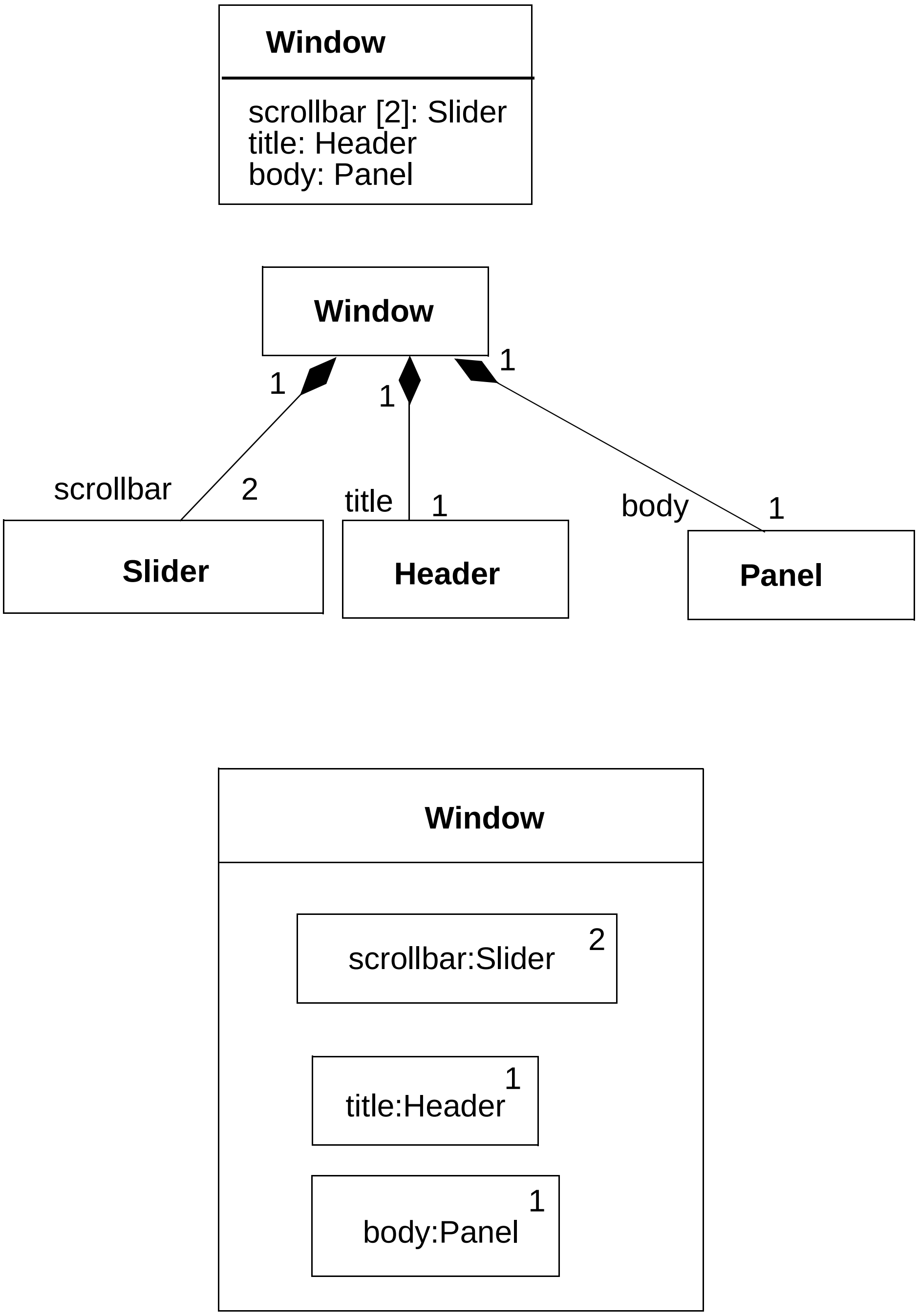


Figure 3-25 Different Ways to Show Composition

3.42.5 Mapping

A class box with an attribute compartment maps into a Class with Attributes. Although attributes may be semantically equivalent to composition on a deep level, the mapped model distinguishes the two forms.

A solid diamond on an association path maps into the composition property on the corresponding Association Role.

A class box with contained class boxes maps into a set of composition associations; that is, one composition association between the Class corresponding to the outer class box and each of the Classes corresponding to the enclosed class boxes. The multiplicity of the composite end of each association is 1. The multiplicity of each constituent end is 1 if not specified explicitly; otherwise, it is the value specified in the corner of the class box *or* specified on an association path from the outer class box boundary to an inner class box.

3.43 Links

3.43.1 Semantics

A link is a tuple (list) of object references. Most commonly, it is a pair of object references. It is an instance of an association.

3.43.2 Notation

A binary link is shown as a path between two objects. In the case of a reflexive association, it may involve a loop with a single object. See “Association” on page 3-52 for details of paths.

A rolename may be shown at each end of the link. An association name may be shown near the path. If present, it is underlined to indicate an instance. Links do not have instance names, they take their identity from the objects that they relate. Multiplicity is *not* shown for links because they are instances. Other association adornments (aggregation, composition, navigation) may be shown on the link roles.

A qualifier may be shown on a link. The value of the qualifier may be shown in its box.

Implementation stereotypes

A stereotype may be attached to the link role to indicate various kinds of implementation. The following stereotypes may be used:

«association»	association (default, unnecessary to specify except for emphasis)
«parameter»	procedure parameter

«local»	local variable of a procedure
«global»	global variable
«self»	self link (the ability of an object to send a message to itself)

N-ary link

An n-ary link is shown as a diamond with a path to each participating object. The other adornments on the association, and the adornments on the roles, have the same possibilities as the binary link.

3.43.3 Example

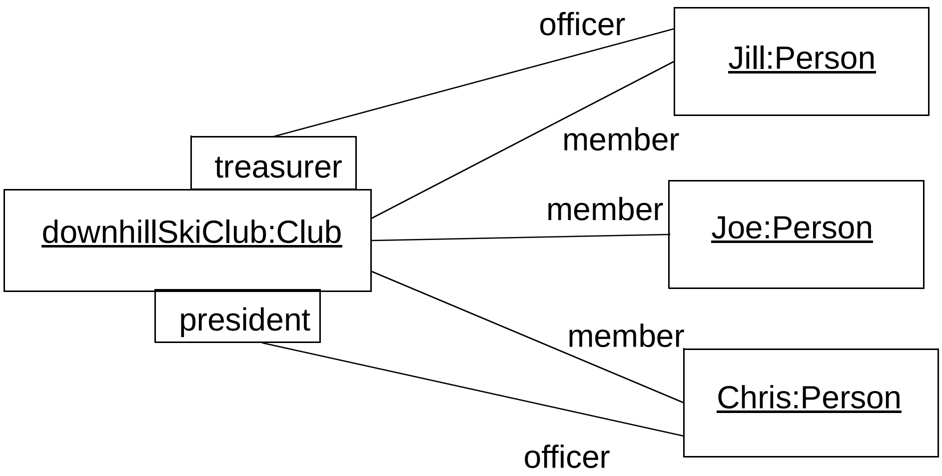


Figure 3-26 Links

3.43.4 Mapping

The mapping depends on the kind of diagram.

- Within a collaboration diagram, each link path maps to an AssociationRole between the ClassifierRoles corresponding to the connected class boxes. If a name is placed on the link path, then it is the name of the Association that is the type of the AssociationRole. Stereotypes on the path indicate the form of the relationship within the collaboration.
- Within an object diagram, each link path maps to a Link between the Objects corresponding to the connected class boxes. If a name is placed on the link path, then it is an instance of the given Association (and the role names must match or the diagram is ill formed).

3.44 Generalization

3.44.1 Semantics

Generalization is the taxonomic relationship between a more general element and a more specific element that is fully consistent with the first element and that adds additional information. It is used for classes, packages, use cases, and other elements.

3.44.2 Notation

Generalization is shown as a solid-line path from the more specific element (such as a subclass) to the more general element (such as a superclass), with a large hollow triangle at the end of the path where it meets the more general element.

A generalization path may have a text label in the following format:

discriminator

where *discriminator* is the name of a partition of the subtypes of the superclass. The subclass is declared to be in the given partition. The absence of a discriminator label indicates the "empty string" discriminator which is a valid value (the "default" discriminator).

Generalization may be applied to associations as well as classes, although the notation may be messy because of the multiple lines. An association can be shown as an association class for the purpose of attaching generalization arrows.

3.44.3 Presentation Options

A group of generalization paths for a given superclass may be shown as a tree with a shared segment (including triangle) to the superclass, branching into multiple paths to each subclass.

If a text label is placed on a generalization triangle shared by several generalization paths to subclasses, the label applies to all of the paths. In other words, all of the subclasses share the given properties.

3.44.4 Details

The existence of additional subclasses in the model that are not shown on a particular diagram may be shown using an ellipsis (. . .) in place of a subclass.

Note – This does not indicate that additional classes may be added in the future. It indicates that additional classes exist right now, but are not being seen. This is a notational convention that information has been suppressed, not a semantic statement.

Predefined constraints may be used to indicate semantic constraints among the subclasses. A comma-separated list of keywords is placed in braces either near the shared triangle (if several paths share a single triangle) or near a dotted line that crosses all of the generalization lines involved. The following keywords (among others) may be used (the following constraints are predefined):

overlapping	A descendent may be descended from more than one subclass.
disjoint	A descendent may not be descended from more than one subclass.
complete	All subclasses have been specified (whether or not shown). No additional subclasses are expected.
incomplete	Some subclasses have been specified, but the list is known to be incomplete. There are additional subclasses that are not yet in the model. This is a statement about the model itself. Note that this is not the same as the ellipsis, which states that additional subclasses exist in the model but are not shown on the current diagram.

The *discriminator* must be unique among the attributes and association roles of the given superclass. Multiple occurrences of the same discriminator name are permitted and indicate that the subclasses belong to the same partition.

The use of multiple classification dynamic classification affects the dynamic execution semantics of the language, but is not unusually apparent from a static model.

3.44.5 Example

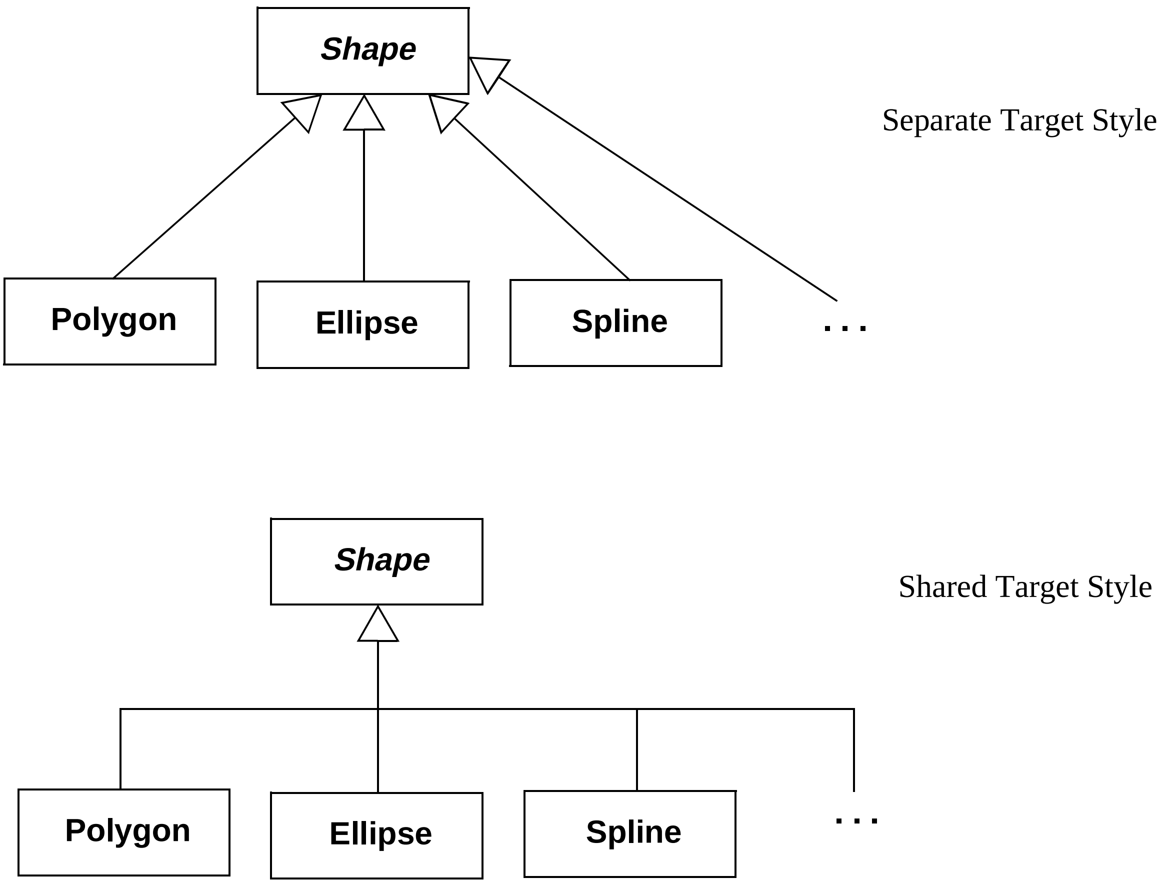


Figure 3-27 Styles of Displaying Generalizations

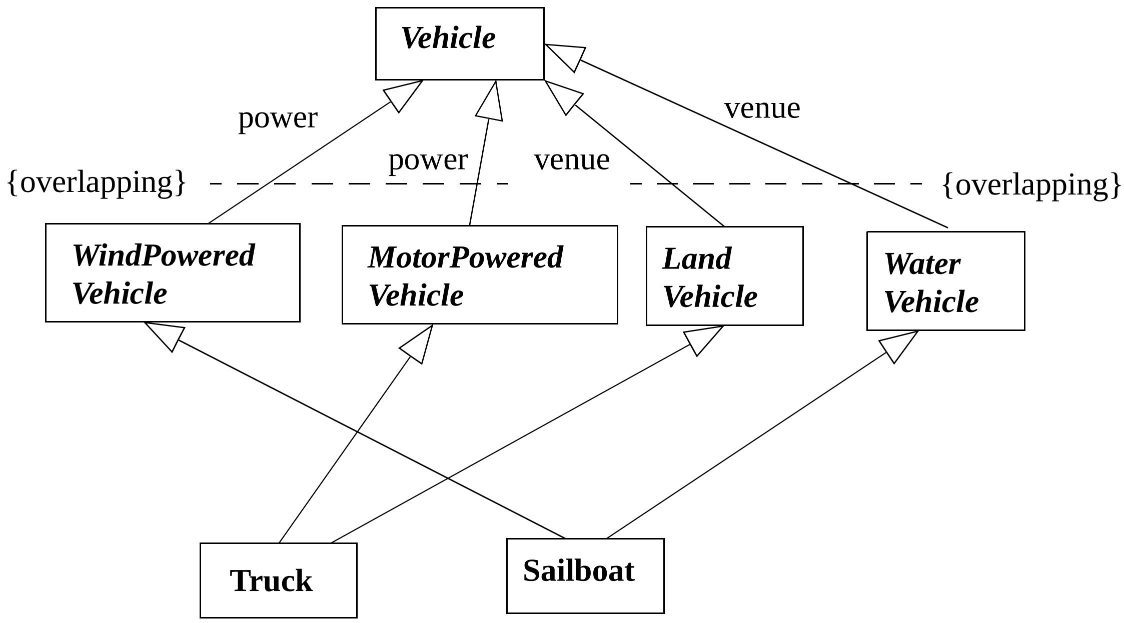


Figure 3-28 Generalization with Discriminators and Constraints, Separate Target Style

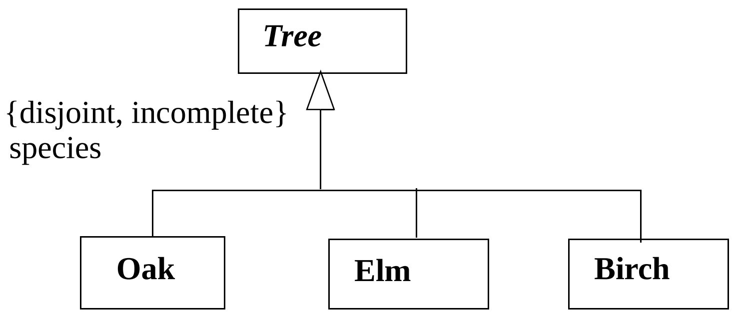


Figure 3-29 Generalization with Shared Target Style

3.44.6 Mapping

Each generalization path between two class boxes maps into a Generalization between the corresponding Classes. A generalization tree with one arrowhead and many tails maps into a set of Generalizations, one between each Class corresponding to a class box on a tail and the single Class corresponding to the class box on the head. That is, a tree is semantically indistinguishable from a set of distinct arrows, it is purely a notational convenience.

Any property string attached to a generalization arrow applies to the Generalization. A property string attached to the head line segment on a generalization tree represents a (duplicated) property on each of the individual Generalizations.

The presence of an ellipsis (“...”) as a subclass node of a given class indicates that the semantic model contains at least one subclass of the given class that is not visible on the current diagram. Normally, this indicator will be maintained automaticallyby an editing tool.

3.45 Dependency

3.45.1 Semantics

A dependency indicates a semantic relationship between two (or more) model elements. It relates the model elements themselves and does not require a set of instances for its meaning. It indicates a situation in which a change to the target element may require a change to the source element in the dependency.

3.45.2 Notation

A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow depends on the model element at the arrowhead. The arrow may be labeled with an optional stereotype and an optional name.

The following kinds of Dependency are predefined and may be indicated with keywords:

trace – Trace:	A historical connection between two elements that represent the same concept at different levels of meaning.
refine – Refinement:	A historical or derivation connection between two elements with a mapping (not necessarily complete) between them. A description of the mapping may be attached to the dependency in a note. Various kinds of refinement have been proposed and can be indicated by further stereotyping.
uses – Usage:	A situation in which one element requires the presence of another element for its correct implementation or functioning. May be stereotyped further to indicate the exact nature of the dependency, such as calling an operation of another class, granting permission for access, instantiating an object of another class, etc.
bind – Binding:	A binding of template parameters to actual values to create a nonparameterized element. See “Part 2 - Diagram Elements” on page 3-6 for more details.

3.45.3 Presentation Options

If one of the elements is a note or constraint, then the arrow may be suppressed because the direction is clear (the note or constraint is the source of the arrow).

3.45.4 Example

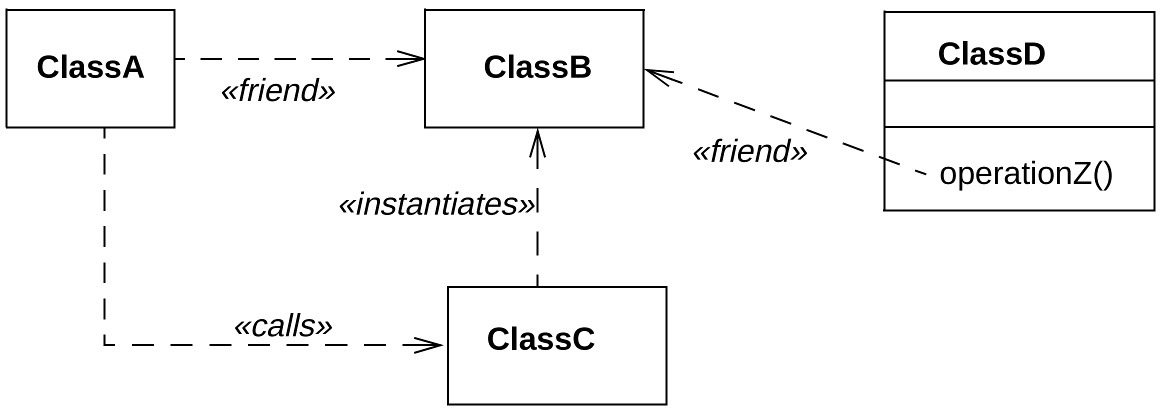


Figure 3-30 Various Usage Dependencies Among Classes

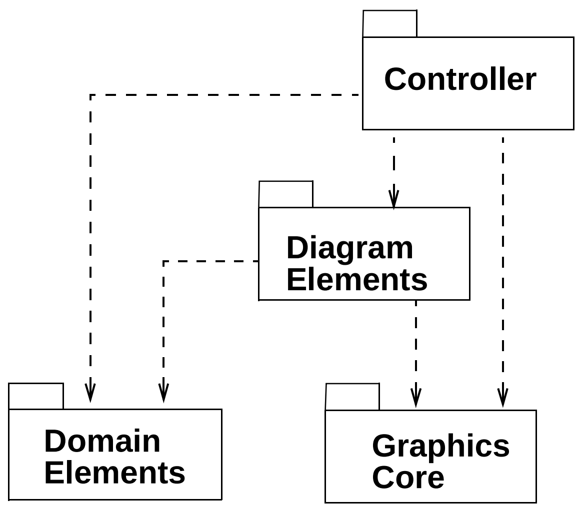


Figure 3-31 Dependencies Among Packages

3.45.5 Mapping

A dashed arrow maps into a Dependency between the Elements corresponding to the symbols attached to the ends of the arrow. The stereotype and the name (if any) attached to the arrow are the stereotype and name of the Dependency.

3.46 Derived Element

3.46.1 Semantics

A derived element is one that can be computed from another one, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

3.46.2 Notation

A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

3.46.3 Style Guidelines

The details of computing a derived element can be specified by a dependency with the stereotype «derived». Usually it is convenient in the notation to suppress the dependency arrow and simply place a constraint string near the derived element, although the arrow can be included when it is helpful.

3.46.4 Example

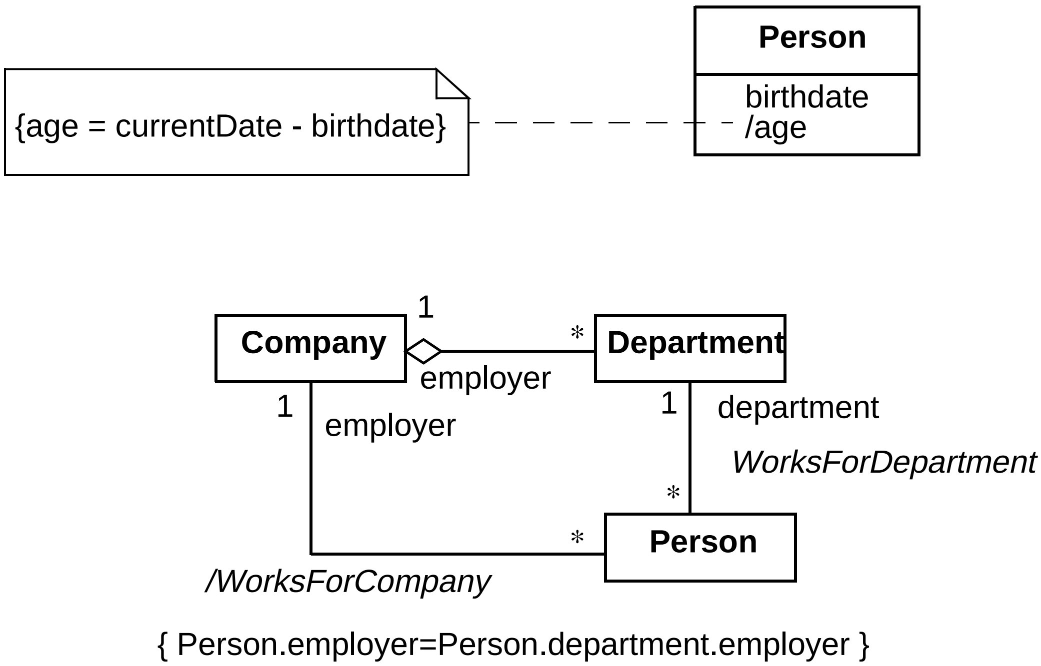


Figure 3-32 Derived Attribute and Derived Association

3.46.5 Mapping

The presence of a derived adornment (a leading “/” on the symbol name) on a symbol maps into the setting of the “derived” property of the corresponding Element.

Part 6 - Use Case Diagrams

A use case diagram shows the relationship among actors and use cases within a system.

3.47 Use Case Diagram

3.47.1 Semantics

Use case diagrams show elements from the use case model. The use case model represents functionality of a system or a class as manifested to external interactors with the system.

3.47.2 Notation

A use case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalizations among the use cases.

3.47.3 Example

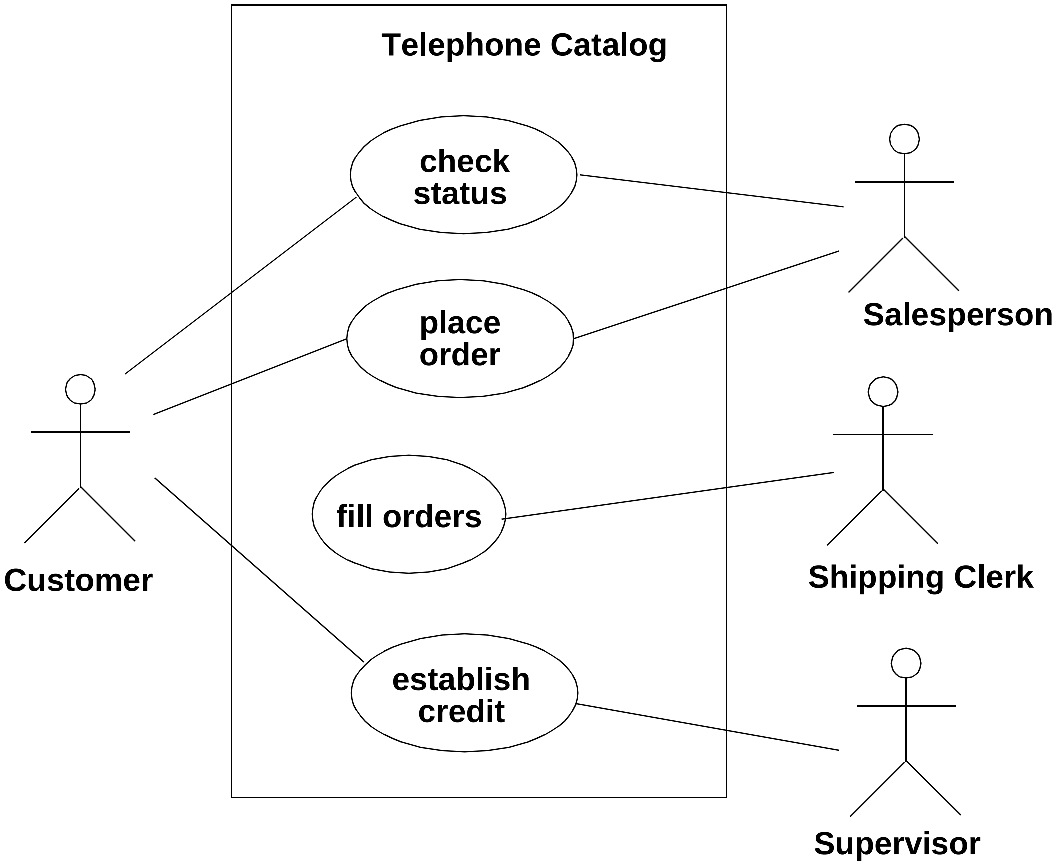


Figure 3-33 Use Case Diagram

3.47.4 Mapping

A set of use case ellipses within a box with connections to actor symbols maps to a single UseCaseModel package containing a set of UseCases and Actors with relationships among them.

3.48 Use Case

3.48.1 Semantics

A use case is a coherent unit of functionality provided by a system or class as manifested by sequences of messages exchanged among the system and one or more outside interactors (called *actors*) together with actions performed by the system.

3.48.2 Notation

A use case is shown as an ellipse containing the name of the use case.

An *extension point* is a location within a use case at which action sequences from other use cases may be inserted. Each extension point must have a unique name within a use case. Extension points may be listed in a compartment of the use case with the heading **extension points**.

3.48.3 Presentation Options

The name of the use case may be placed below the ellipse.

3.48.4 Style Guidelines

Use case names should follow capitalization and punctuation guidelines used for behavioral items in the same model.

3.48.5 Mapping

A use case symbol maps to a UseCase with the given name (if any). An extension point maps into an ExtensionPoint within the UseCase.

3.49 Actor

3.49.1 Semantics

An actor is a role of object or objects outside of a system that interacts directly with it as part of a coherent work unit (a use case). An Actor element characterizes the role played by an outside object. One physical object may play several roles; therefore, it may be modeled by several actors.

3.49.2 Notation

An actor may be shown as a class rectangle with the stereotype “actor.” The standard stereotype icon for an actor is the “stick man” figure with the name of the actor below the figure.

3.49.3 Style Guidelines

Actor names should follow capitalization and punctuation guidelines used for types and classes in the same model.

3.49.4 Mapping

An actor symbol maps to an Actor with the given name.

3.50 Use Case Relationships

3.50.1 Semantics

There are several standard relationships among use cases or between actors and use cases.

- Communicates – The participation of an actor in a use case. This is the only relationship between actors and use cases.
- Extends – An extends relationships from use case A to use case B indicates that an instance of use case B may include (subject to specific conditions specified in the extension) the behavior specified by A. Behavior specified by several extenders of a single target use case may occur within a single use case instance.
- Uses – A uses relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B.

3.50.2 Notation

The communication relationship between an actor and a use case is shown as a solid line between the actor and the use case.

An “extends” relationship between use cases is shown by a generalization arrow from the use case providing the extension to the base use case. The arrow is labeled with the stereotype «extends».

A “uses” relationship between use cases is shown by a generalization arrow from the use case doing the use to the use case being used. The arrow is labeled with the stereotype «uses».

The relationship between a use case and its external interaction sequences are usually shown by an invisible hyperlink to sequence diagrams. The relationship between a use case and its implementation may be shown as a refinement relationship to a collaboration, but may also be shown as an invisible hyperlink. The expectation is that a tool will support the ability to “zoom into” a use case to see its scenarios and/or implementation as an interaction.

3.50.3 Example

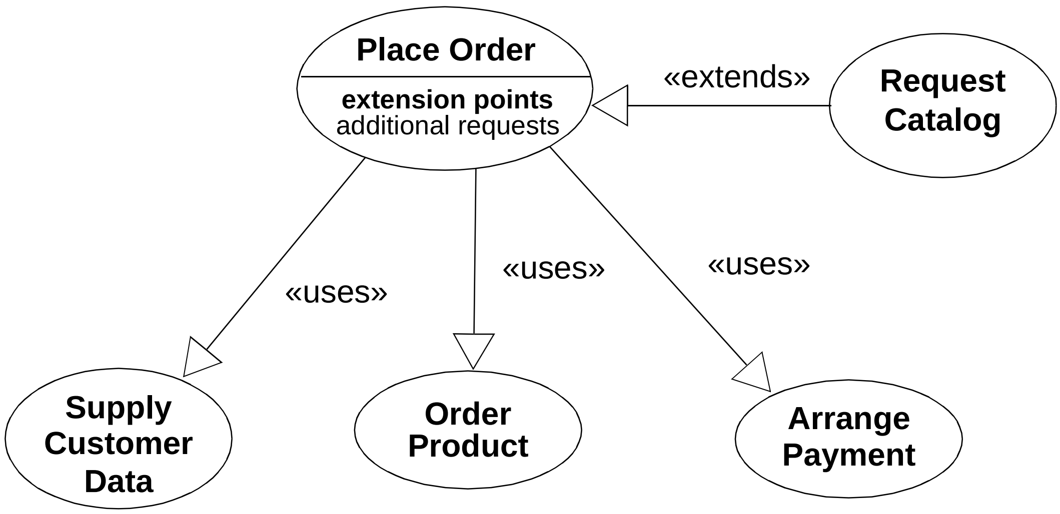


Figure 3-34 Use Case Relationships

3.50.4 Mapping

A path between use case and/or actor symbols maps into the corresponding relationship between the corresponding Elements, as described above.

Part 7 - Sequence Diagrams

3.51 Kinds of Interaction Diagrams

A pattern of interaction among objects is shown on an interaction diagram. Interaction diagrams come in two forms based on the same underlying information, but each emphasizing a particular aspect of it. The two forms are: sequence diagrams and collaboration diagrams.

A *sequence diagram* shows an interaction arranged in time sequence. In particular, it shows the objects participating in the interaction by their “lifelines” and the messages that they exchange arranged in time sequence. It does not show the associations among the objects.

Sequence diagrams come in several slightly different formats intended for different purposes.

A sequence diagram can exist in a generic form (describes all the possible sequences) and in an instance form (describes one actual sequence consistent with the generic form). In cases without loops or branches, the two forms are isomorphic.

Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for complex scenarios. Collaboration diagrams show the relationships among objects and are better for understanding all of the effects on a given object and for procedural design.

3.52 Sequence Diagram

3.52.1 Semantics

A sequence diagram represents an Interaction, which is a set of messages exchanged among objects within a collaboration to effect a desired operation or result.

3.52.2 Notation

A sequence diagram has two dimensions: 1) the vertical dimension represents time and 2) the horizontal dimension represents different objects. Normally time proceeds down the page. (The dimensions may be reversed, if desired.) Usually only time sequences are important, but in real-time applications the time axis could be an actual metric. There is no significance to the horizontal ordering of the objects. Objects can be grouped into “swimlanes” on a diagram.

See subsequent sections for details of the contents of a sequence diagram.

Note that much of this notation is drawn directly from the Object Message Sequence Chart notation of Buschmann, Meunier, Rohnert, Sommerlad, and Stal, which is itself derived with modifications from the Message Sequence Chart notation.

3.52.3 Presentation Options

The horizontal ordering of the lifelines is arbitrary. Often call arrows are arranged to proceed in one direction across the page; however, this is not always possible and the ordering does not convey information.

The axes can be interchanged, so that time proceeds horizontally to the right and different objects are shown as horizontal lines.

Various labels (such as timing marks, descriptions of actions during an activation, and so on) can be shown either in the margin or near the transitions or activations that they label.

3.52.4 Example

Simple sequence diagram with concurrent objects

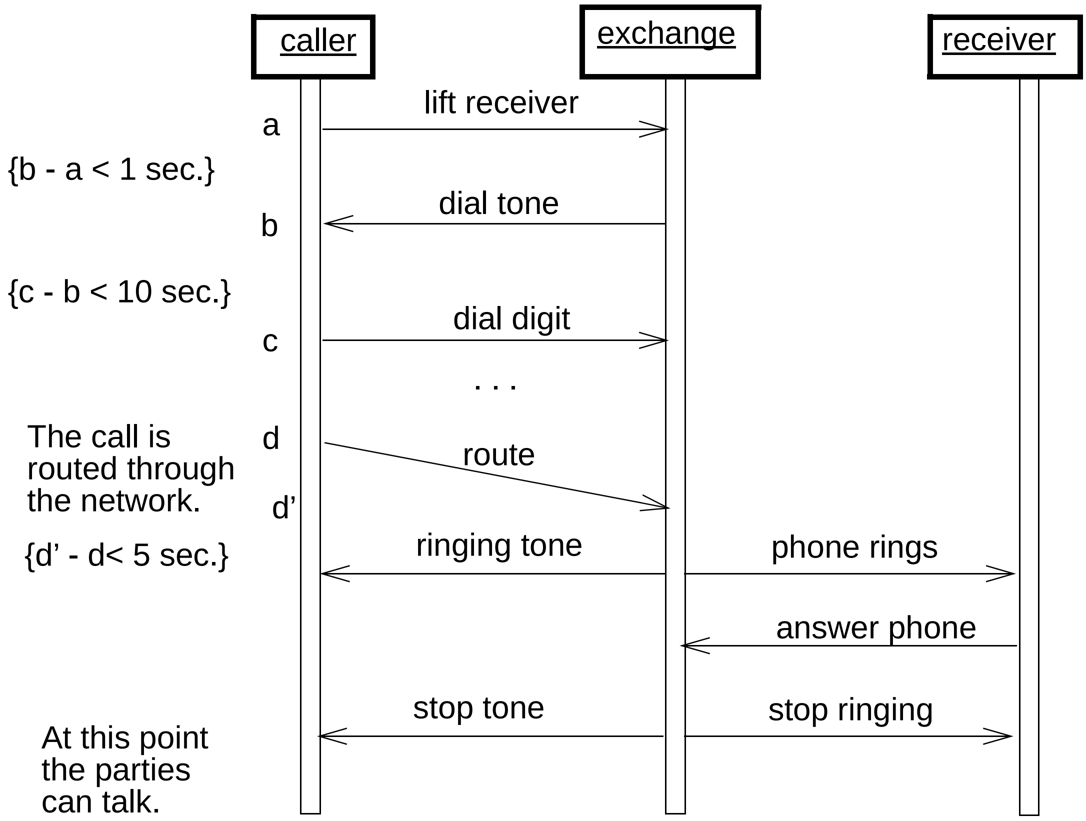


Figure 3-35 Simple Sequence Diagram with Concurrent Objects

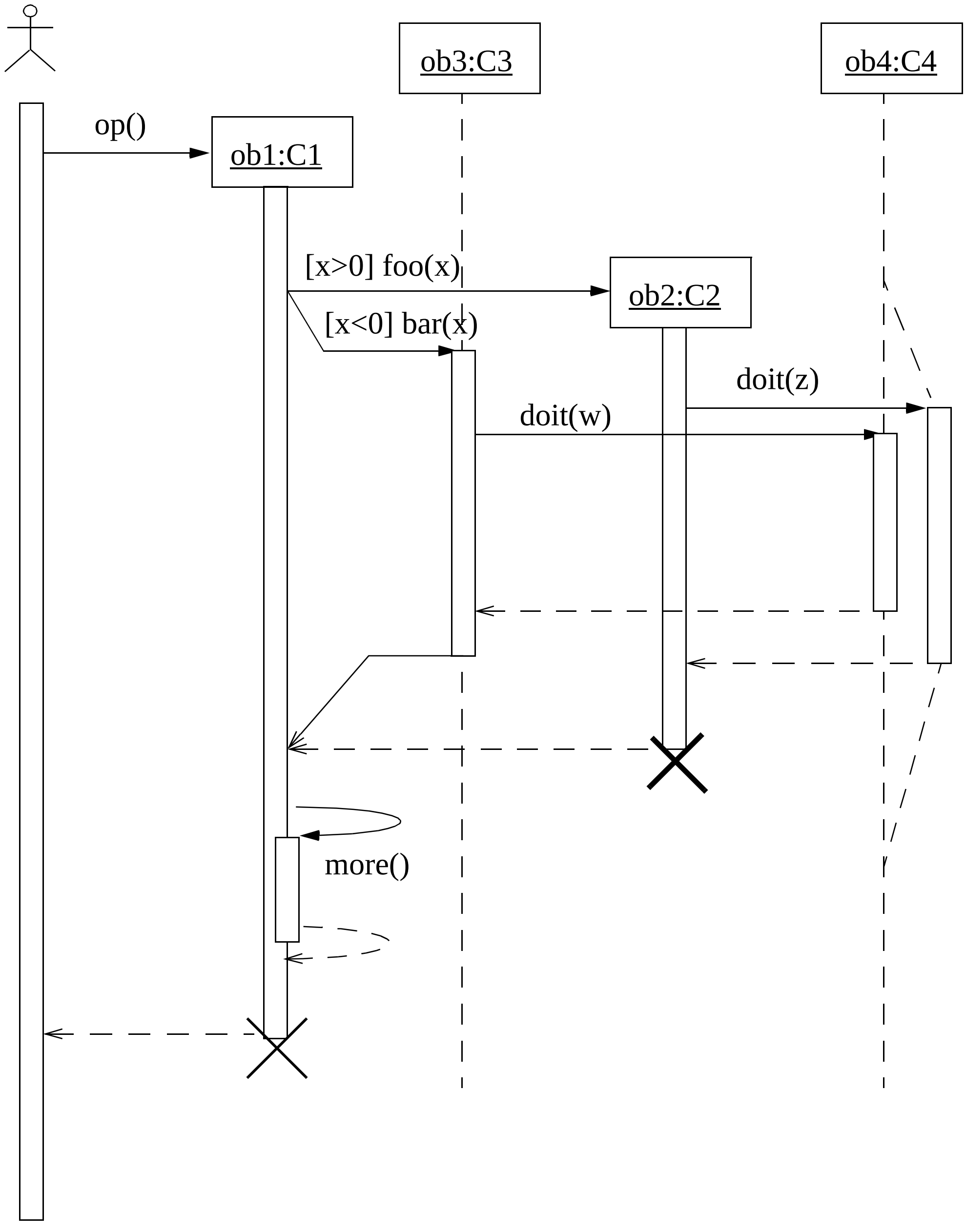


Figure 3-36 Sequence Diagram with Focus of Control, Conditional, Recursion, Creation, Destruction

3.52.5 Mapping

This section summarizes the mapping for the sequence diagram and the elements within it, some of which are described in subsequent sections.

Sequence diagram

A sequence diagram maps into an Interaction and an underlying Collaboration. Each object box with its lifeline maps into a ClassifierRole. The name field maps into the ClassifierRole name and the type field maps into the *type* association from the role to the Classifier with the given name. The associations among roles are not shown on the sequence diagram. They must be obtained in the model from a complementary collaboration diagram or other means. A message arrow maps into a Message between the ClassifierRoles corresponding to the two lifelines that the arrow connects. Unless the correct AssociationRole can be determined from a complementary collaboration diagram or other means, the Message must be attached to a dummy AssociationRole implied between the two ClassifierRoles for lack of complete information. A timing label placed on the level of an arrow endpoint maps into the name of the corresponding Message. A constraint placed on the diagram maps into a Constraint on the entire Interaction.

An object symbol placed within the frame of the diagram maps into a CreateAction attached to the Message corresponding to the incoming arrow. If an object termination symbol (“X”) is the target of an arrow, it maps into a DestroyAction attached to the Message corresponding to the arrow; otherwise, it maps into a TerminateAction.

On a diagram with concurrent objects, a *predecessor* association is established between Messages corresponding to successive arrows in the vertical sequence. In case of concurrent arrows, the mapping to a *predecessor* sequence may be ambiguous and may require additional information.

Procedural sequence diagram

On a procedural sequence diagram (one with focus of control and calls), subsequent arrows on the same lifeline map into Messages obeying the *predecessor* association. An arrow to the head of a focus of control region establishes a nested activation. It maps into a Message (synchronous, activation) with associated CallAction (holding the arguments and referencing the target Operation between the ClassifierRoles corresponding to the lifelines. All arrows departing the nested activation map into Messages with an *activation* Association to the Message corresponding to the arrow at the head of the activation. A return arrow departing the end of the activation maps into a Message (synchronous, reply) with:

- an *activation* Association to the Message corresponding to the arrow at the head of the activation, and
- a *predecessor* association to the previous message within the same activation.

A return must be the final message within a predecessor chain. It is not the predecessor of any message. Any guard conditions or iteration conditions attached to a message arrow become *recurrence* values of the Message. The operation name is used to select the target Operation with the given name. The operation arguments become *argument* Expressions on the Action.

3.53 Object Lifeline

3.53.1 Semantics

A Role is a slot for an object within a collaboration that describes the type of object that may play the role and describes its relationships to other Roles. Within a sequence diagram the existence and duration of the object in a role is shown, but the relationships among the roles is not shown. There are ClassifierRoles and AssociationRoles.

3.53.2 Notation

An object role is shown as a vertical dashed line called the “lifeline.” The lifeline represents the existence of the object at a particular time. If the object is created or destroyed during the period of time shown on the diagram, then its lifeline starts or stops at the appropriate point; otherwise, it goes from the top to the bottom of the diagram. An object symbol is drawn at the head of the lifeline. If the object is created during the diagram, then the message that creates it is drawn with its arrowhead on the object symbol. If the object is destroyed during the diagram, then its destruction is marked by a large “X,” either at the message that causes the destruction or (in the case of self-destruction) at the final return message from the destroyed object. An object that exists when the transaction starts is shown at the top of the diagram (above the first arrow). An object that exists when the transaction finishes has its lifeline continue beyond the final arrow.

The lifeline may split into two or more concurrent lifelines to show conditionality. Each separate track corresponds to a conditional branch in the message flow. The lifelines may merge together at some subsequent point.

3.53.3 Example

See Figure 3-36 on page 3-84.

3.53.4 Mapping

See “Mapping” on page 3-85.

3.54 Activation

3.54.1 Semantics

An activation (focus of control) shows the period during which an object is performing an action either directly or through a subordinate procedure. It represents both the duration of the action in time and the control relationship between the activation and its callers (stack frame).

3.54.2 Notation

An activation is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time. The action being performed may be labeled in text next to the activation symbol or in the left margin, depending on style. Alternately, the incoming message may indicate the action, in which case it may be omitted on the activation itself. In procedural flow of control, the top of the activation symbol is at the tip of an incoming message (the one that initiates the action) and the base of the symbol is at the tail of a return message.

In the case of concurrent objects each with their own threads of control, an activation shows the duration when each object is performing an operation. Operations by other objects are not relevant. If the distinction between direct computation and indirect computation (by a nested procedure) is unimportant, the entire lifeline may be shown as an activation.

In the case of procedural code, an activation shows the duration during which a procedure is active in the object or a subordinate procedure is active, possibly in some other object. In other words, all of the active nested procedure activations may be seen at a given time. In the case of a recursive call to an object with an existing activation, the second activation symbol is drawn slightly to the right of the first one, so that they appear to "stack up" visually. (Recursive calls may be nested to an arbitrary depth.)

3.54.3 Example

See Figure 3-36 on page 3-84.

3.54.4 Mapping

See "Mapping" on page 3-85.

3.55 Message

3.55.1 Semantics

A message is a communication between objects that conveys information with the expectation that action will ensue. The receipt of a message is one kind of event.

3.55.2 Notation

A message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. In case of a message from an object to itself, the arrow may start and finish on the same object symbol. The arrow is labeled with the name of the message (operation or signal) and its argument values. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often omitted in sequence diagrams, in which the physical location of the arrow shows the relative sequences, but they are necessary in collaboration diagrams. Sequence numbers are useful on both kinds of diagrams for identifying concurrent threads of control. A message may also be labeled with a guard condition.

3.55.3 Presentation options

Variation: Asynchronous

An asynchronous message is drawn with a half-arrowhead (one with only one wing instead of two).

Variation: Call

A procedure call is drawn as a full arrowhead. A return is shown as a dashed arrow.

Variation:

In a procedural flow of control, the return arrow may be omitted (it is implicit at the end of an activation). It is assumed that every call has a paired return after any subordinate messages. The return value can be shown on the initial message line. For nonprocedural flow of control (including parallel processing and asynchronous messages) returns should be shown explicitly.

Variation:

In a concurrent system, a full arrowhead shows the yielding of a thread of control (wait semantics) and a half arrowhead shows the sending of a message without yielding control (no-wait semantics).

Variation:

Normally message arrows are drawn horizontally. This indicates the duration required to send the message is “atomic,” that is, it is brief compared to the granularity of the interaction and that nothing else can “happen” during the message transmission. This is the correct assumption within many computers. If the message requires some time to arrive, during which something else can occur (such as a message in the opposite direction), then the message arrow may be slanted downward so that the arrowhead is below the arrow tail.

Variation: Branching

A branch is shown by multiple arrows leaving a single point, each labeled by a guard condition. Depending on whether the guard conditions are mutually exclusive, the construct may represent conditionality or concurrency.

Variation: Iteration

A connected set of messages may be enclosed and marked as an iteration. For a scenario, the iteration indicates that the set of messages can occur multiple times. For a procedure, the continuation condition for the iteration may be specified at the bottom of the iteration. If there is concurrency, then some messages in the diagram may be part of the iteration and others may be single execution. It is desirable to arrange a diagram so that the messages in the iteration can be enclosed together easily.

Variation:

A lifeline may subsume an entire set of objects on a diagram representing a high-level view.

Variation:

A distinction may be made between a period during which an object has a live activation and a period in which the activation is actually computing. The former (during which it has control information on a stack but during which control resides in something that it called) is shown with the ordinary double line. The latter (during which it is the top item on the stack) may be distinguished by shading the region.

3.55.4 Mapping

See “Mapping” on page 3-85.

3.56 Transition Times

3.56.1 Semantics

A message may have a sending time and a receiving time. These are formal names that may be used within constraint expressions. The two may be the same (if the message is considered atomic) or different (if its delivery is nonatomic).

3.56.2 Notation

A transition instance (such as a message in a sequence diagram, a collaboration diagram, or a transition in a state machine) may be given a name. The name represents the time at which a message is sent (example: A). In cases where the delivery of the message in not instantaneous, the time at which the message is received is indicated by the transition name with a prime sign appended (example: A'). The name may be shown in the left margin aligned with the arrow (on a sequence diagram) or near the tail of the message flow arrow (on a collaboration diagram). This name may be used in constraint expressions to designate the time the message was sent. If the message line is slanted, then the primed-name indicates the time at which the message is received.

Constraints may be specified by placing Boolean expressions in braces on the sequence diagram.

3.56.3 Example

See Figure 3-35 on page 3-83.

3.56.4 Mapping

See “Mapping” on page 3-85.

Part 8 - Collaboration Diagrams

A collaboration diagram shows an interaction organized around the objects in the interaction and their links to each other. Unlike a sequence diagram, a collaboration diagram shows the relationships among the object roles. On the other hand, a collaboration diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads must be determined using sequence numbers.

3.57 Collaboration

3.57.1 Semantics

Behavior is implemented by sets of objects that exchange messages within an overall interaction to accomplish a purpose. To understand the mechanisms used in a design, it is important to see only the objects and the messages involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are part for other purposes. Such a static construct is called a *collaboration*.

A collaboration is a set of participants and relationships that are meaningful for a given set of purposes. The identification of participants and their relationships does not have global meaning.

A collaboration may be attached to an operation or a use case to describe the context in which their behavior occurs. The actual behavior may be specified in interactions, such as sequence diagrams or collaboration diagrams. A collaboration may also be attached to a class to define the class’s static structure.

A parameterized collaboration represents a design construct that can be used repeatedly in different designs. The participants in the collaboration, including the classes and relationships, can be parameters of the generic collaboration. The parameters are bound to particular model elements in each instantiation of generic collaboration. Such a parameterized collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most collaborations can be anonymous because they are attached to a named entity, patterns are free standing design constructs that must have names.

A collaboration may be expressed at different levels of granularity. A coarse-grained collaboration may be refined to produce another collaboration that has a finer granularity.

3.57.2 Notation

The description of behavior involves two aspects: 1) the structural description of its participants and 2) the behavioral description of its execution. The two aspects are often described together on a single diagram, but at times it is useful to describe the structural and behavioral aspects separately. The structure of objects playing roles in a behavior and their relationships is called a *collaboration.* A collaboration shows the context in which interaction occurs. The dynamic behavior of the message sequences exchanged among objects to accomplish a specific purpose is called an *interaction.* A collaboration is shown by a collaboration diagram without messages. By adding messages, an interaction is shown. Different sets of messages may be applied to the same collaboration to yield different interactions.

3.58 Collaboration Diagram

3.58.1 Semantics

A collaboration diagram represents a Collaboration, which is a set of objects related in a particular context, and an Interaction, which is a set of messages exchanged among the objects within a collaboration to effect a desired operation or result.

3.58.2 Notation

A collaboration diagram is a graph of references to objects and links with message flows attached to its links. The diagram shows the objects relevant to the performance of an operation, including objects indirectly affected or accessed during the operation. The collaboration used to describe an operation includes its arguments and local variables created during its execution as well as ordinary associations.

- Objects created during the execution may be designated as {new}.
- Objects destroyed during the execution may be designated as {destroyed}.
- Objects created during the execution and then destroyed may be designated as {transient}.

These changes in life state are derivable from the detailed messages sent among the objects, they are provided as notational conveniences.

The diagram also shows the links among the objects, including transient links representing procedure arguments, local variables, and *self* links. Because collaboration diagrams often are used to help design procedures, they typically show navigability using arrowheads on links. (An arrowhead on a line between object boxes indicates a link with one-way navigability. An arrow next to a line indicates a message flowing in the given direction over the link. Obviously a message arrow cannot flow backwards over a one-way link.)

Individual attribute values are usually not shown explicitly. If messages must be sent to attribute values, the attributes should be modeled using associations instead.

The internal messages that implement a method are numbered starting with number 1. For a procedural flow of control, the subsequent message numbers are nested in accordance with call nesting. For a nonprocedural sequence of messages exchanged among concurrent objects, all the sequence numbers are at the same level (that is, they are not nested).

A collaboration diagram without messages shows the *context* in which interactions can occur, without showing any specific interactions. It might be used to show the context for a single operation or even for all of the operations of a class or group of classes.

3.58.3 Example

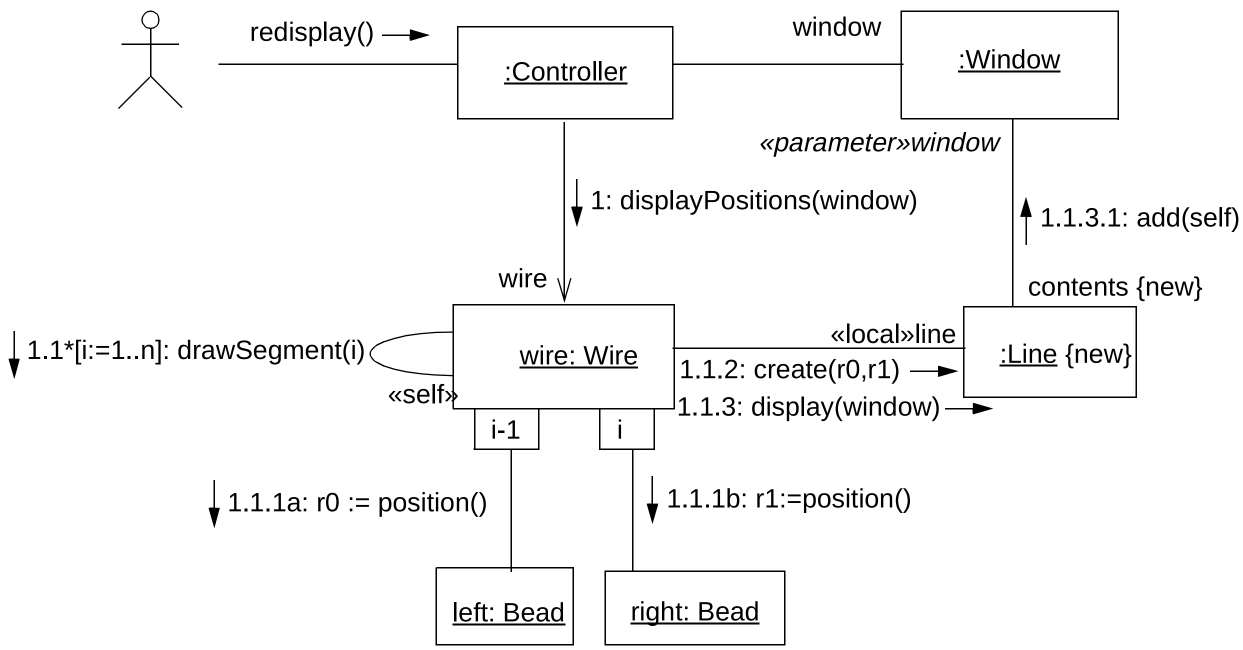


Figure 3-37 Collaboration Diagram

3.58.4 Mapping

A collaboration diagram maps to a Collaboration with a superimposed Interaction.

3.59 Pattern Structure

3.59.1 Semantics

A collaboration can be used to specify the implementation of design constructs. For this purpose, it is necessary to specify its context and interactions. It is also possible to view a collaboration as a single entity from the “outside.” For example, this could be used to identify the presence of design patterns within a system design. A pattern is a parameterized collaboration. In each use of the pattern, actual classes are substituted for the parameters in the pattern definition.

Note that *patterns* as defined in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides include much more than structural descriptions. UML describes the structural aspects and some behavioral aspects of design patterns; however, UML notation does not include other important aspects of patterns, such as usage trade-offs or examples. These must be expressed in text or tables.

3.59.2 Notation

A use of a collaboration is shown as a dashed ellipse containing the name of the collaboration. A dashed line is drawn from the collaboration symbol to each of the objects or classes (depending on whether it appears within an object diagram or a class diagram) that participate in the collaboration. Each line is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the collaboration; such names in the collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. Therefore, a collaboration symbol can show the use of a design pattern together with the actual classes that occur in that particular use of the pattern.

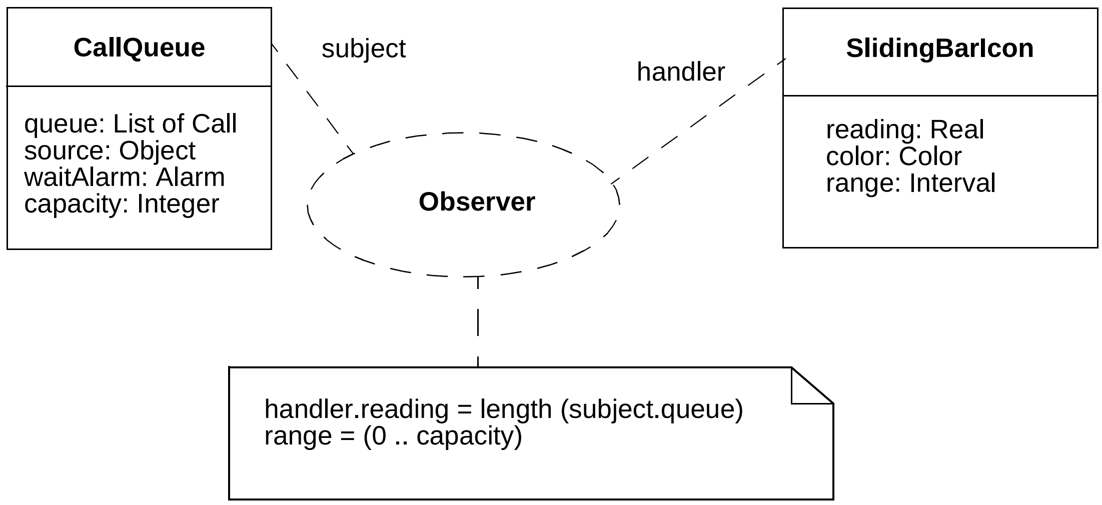


Figure 3-38 Use of a Collaboration

3.59.3 Mapping

A collaboration usage symbol maps into a Collaboration. For each class symbol attached by an arrow to the pattern occurrence symbol, the corresponding Class is bound to the template parameter that is the *type* association target of the ClassifierRole in the Pattern with the name equal to the name on the arrow.

3.60 Collaboration Contents

The contents of a collaboration are modeling elements that interact within a given context for a particular purpose, such as performing an operation or a use case, it is a "society of objects." A collaboration is a fragment of a larger complete model that is intended for a particular purpose.

3.60.1 Semantics

A *collaboration* shows one or more roles together with their contents, associations, and neighbor roles, plus additional relationships and classes as needed. To use a collaboration, each role must be bound to an actual class that can support the operations required of the role.

3.60.2 Notation

A collaboration is shown as a graph of class references and association references. Each reference is a *role* of the collaboration; that is, each entity is playing a role within the context of the collaboration, a role that is only part of its full description. The names of the objects represent their roles within the collaboration. A collaboration is a prototype; in each use of the collaboration the roles are bound to actual objects. There are several ways to show the diagram:

Methods

If the collaboration shows the implementation of an operation (a method), then it is usually drawn as a separate collaboration diagram including context to which message flow is added to obtain an interaction. The collaboration for the operation includes the target object of the operation and any other objects that it calls on, directly or indirectly, to implement the operation. The collaboration includes the objects present before the operation, the objects present after the operation (these may be the same or mostly the same as the ones before), and objects that exist only during the operation (these may be marked as «new», «destroyed», and «transient»). Only objects involved in the operation implementation need to be shown. To show the implementation of an operation, message flows are superimposed on the links between objects in the collaboration; each flow shows a step within the method for the operation (see “Message flows” on page 3-101).

Classes

A collaboration is normally defined for a single operation. By taking the union of all of the collaborations for all of the operations of a class, an overall collaboration for the entire class can be shown. This collaboration shows all of the context for the implementation of the class.

In both cases, the usual assumption is that objects and classes not shown on the collaboration are not affected by the operation. It is not always safe to assume that all of the objects on a collaboration diagram *are* used by the operation.

Different collaborations may be devised for the same class for different purposes. Each collaboration may show a somewhat different subset of attributes, operators, and related objects that are relevant to each purpose. Where actual operations often fall into related groups, each collaboration might specify a consistent view shared by several operations that is somewhat different from the view needed by other operations on the same type. Similarly, the model of types in a business organization can often be divided into several collaborations, each from the point of view of a particular stakeholder.

3.61 Interactions

A collaboration of objects interacts to accomplish a purpose (such as performing an operation) by exchanging messages. The messages may include both signals and calls, as well as more implicit interaction through conditions and time events. A specific pattern of message exchanges to accomplish a specific purpose is called an *interaction*.

3.61.1 Semantics

An *interaction* is a behavioral specification that comprises a sequence of message exchanges among a set of objects within a collaboration to accomplish a specific purpose, such as the implementation of an operation. To specify an interaction, it is first necessary to specify a collaboration; that is, to establish the objects that interact and their relationships. Then the possible interaction sequences are specified. These can be specified in a single description containing conditionals (branches or conditional signals), or they can be specified by supplying multiple descriptions, each describing a particular path through the possible execution paths.

3.61.2 Notation

Interactions are shown as sequence diagrams or as collaboration diagrams. Both diagram formats show the execution of collaborations. However, sequence diagrams only show the participating objects and do not show their relationships to other objects or their attributes; therefore, they do not fully show the context aspect of a collaboration. Sequence diagrams do show the behavioral aspect of collaborations explicitly, including the time sequence of message and explicit representation of method activations. Sequence diagrams are described in “Sequence Diagram” on page 3-82. Collaboration diagrams show the full context of an interaction, including the objects and their relationships relevant to a particular interaction, so they are often better for design purposes. Collaboration diagrams are described in the following sections.

3.61.3 Example

See Collaboration Diagram section for a collaboration underlying an interaction.

3.62 Collaboration Roles

3.62.1 Semantics

A Role is a slot for an object within a collaboration that describes the type of object that may play the role and describes its relationships to other Roles. There are ClassifierRoles and AssociationRoles.

3.62.2 Notation

A collaboration role is shown using the notation for an object or a link. Keep in mind, however, that in the context of a collaboration these represent roles that *bind* to actual objects or links when the collaboration is used, not actual objects and links.

A class role is shown as a class rectangle symbol. Normally only the name compartment is shown. The name compartment contains the string:

classRoleName : *Classifiername*

The classname can include a full pathname of enclosing packages, if necessary. A tool will normally permit shortened pathnames to be used when they are unambiguous. The package names precede the classname and are separated by double colons. For example:

```
display_window: WindowingSystem::GraphicWindows::Window
```

A stereotype for the class may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. The stereotype for an object must match the stereotype for its class.

A class role representing a set of objects includes a multiplicity indicator (such as "*") in the upper right corner of the class box.

An association role is shown as a path between two class role symbols. If the name of the corresponding association is included, it is underlined. Rolenames are not underlined. Even in absence of underlining, a line connecting class roles is an association role.

If one end of the association role path is connected to a multiple class role, then a multiplicity indicator may be placed on that end to emphasize the multiplicity.

3.62.3 Presentation options

The name of the object may be omitted. In this case, the colon should be kept with the class name. This represents an anonymous object of the given class given identity by its relationships.

The class of the object may be suppressed (together with the colon).

3.62.4 Example

See Figure 3-37 on page 3-92.

3.62.5 Mapping

The object symbol in a collaboration diagram maps to a ClassifierRole whose name matches the *object* part of the name string; the role has a *type* Association to a Classifier whose name matches the *type* part of the name string.

3.63 Multiobject

3.63.1 Semantics

A multi-object represents a set of objects on the “many” end of an association. This is used to show operations that address the entire set, rather than a single object in it. The underlying static model is unaffected by this grouping. This corresponds to an association with multiplicity “many” used to access a set of associated objects.

3.63.2 Notation

A multi-object is shown as two rectangles in which the top rectangle is shifted slightly vertically and horizontally to suggest a stack of rectangles. A message arrow to the multi-object symbol indicates a message to the set of objects (for example, a selection operation to find an individual object).

To perform an operation on each object in a set of associated objects requires two messages: 1) an iteration to the multi-object to extract links to the individual objects and then 2) a message sent to each individual object using the (temporary) link. This may be elided on a diagram by combining the messages into a single message that includes an iteration and an application to each individual object. The target rolename takes a “many” indicator (*) to show that many individual links are implied. Although this may be written as a single message, in the underlying model (and in any actual code) it requires the two layers of structure (iteration to find links, message using each link) mentioned previously.

An object from the set is shown as a normal object symbol, but it may be attached to the multi-object symbol using a composition link to indicate that it is part of the set. A message arrow to the simple object symbol indicates a message to an individual object.

Typically a selection message to a multi-object returns a reference to an individual object, to which the original sender then sends a message.

3.63.3 Example

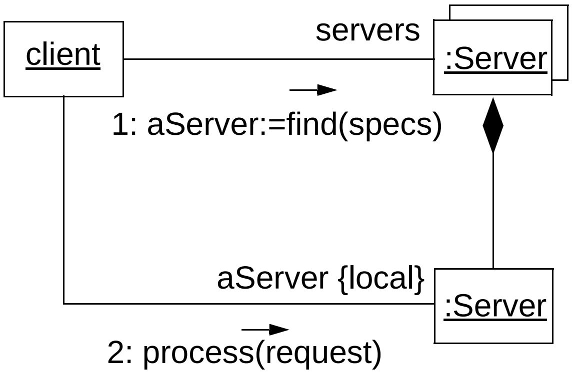


Figure 3-39 Multi-object

3.63.4 Mapping

A multi-object symbol maps to a ClassifierRole with multiplicity "many" (or whatever is explicitly specified). In other respects, it maps the same as an object symbol.

3.64 Active object

An active object is one that owns a thread of control and may initiate control activity. A passive object is one that holds data, but does not initiate control. However, a passive object may send messages in the process of processing a request that it has received. In a collaboration diagram, a ClassifierRole that is an active class represents the active objects that occur during execution.

3.64.1 Semantics

An active object is an object that owns a thread of control. Processes and tasks are traditional kinds of active objects.

3.64.2 Notation

A role for an active object is shown as an object symbol with a heavy border. Frequently active object roles are shown as composites with embedded parts.

The property keyword *{active}* may also be used to indicate an active object.

3.64.3 Example

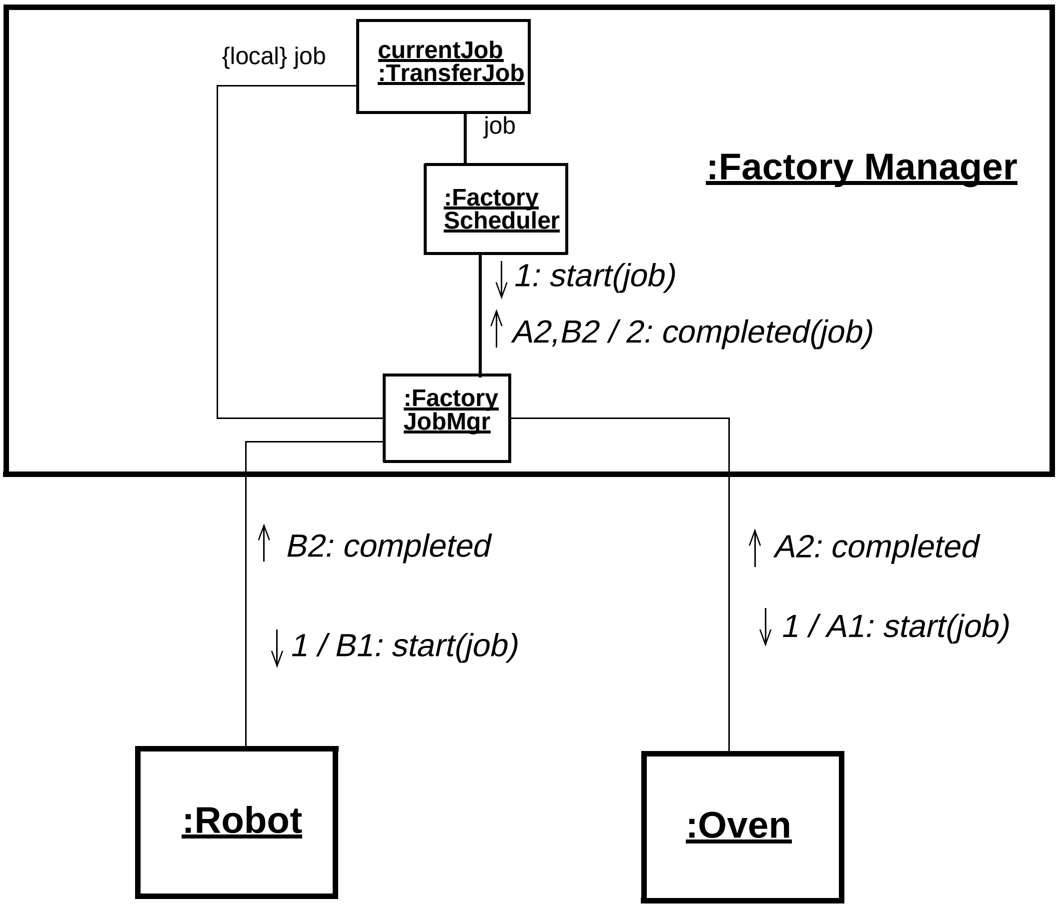


Figure 3-40 Composite Active Object

3.64.4 Mapping

An active object symbol maps as an object symbol does, with the addition that the *active* property is set.

A nested object symbol (active or not) maps into a Classifierrole that has a composition association to the roles corresponding to its contents, as described under Composition.

3.65 Message flows

3.65.1 Semantics

A *message flow* is the sending of a message from one object to another. The implementation of a message may take various forms, such as a procedure call, the sending of a signal between active threads, the explicit raising of events, and so on.

3.65.2 Notation

A message flow is shown as a labeled arrow placed near a link. The meaning is that the link is used to transport, or otherwise implement, the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message).

Control flow type

The following arrowhead variations may be used to show different kinds of messages:

filled solid arrowhead

Procedure call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. May be used with ordinary procedure calls. May also be used with concurrently active objects when one of them sends a signal and waits for a nested sequence of behavior to complete.

stick arrowhead

Flat flow of control. Each arrow shows the progression to the next step in sequence. Normally all of the messages are asynchronous.

half stick arrowhead

Asynchronous flow of control. Used instead of the stick arrowhead to explicitly show an asynchronous message between two objects in a procedural sequence.

other variations

Other kinds of control may be shown, such as “balking” or “time-out;” however, these are treated as extensions to the UML core.

Message label

The label has the following syntax:

predecessor guard-condition sequence-expression return-value := *message-name argument-list*

The label indicates the message sent, its arguments and return values, and the sequencing of the message within the larger interaction, including call nesting, iteration, branching, concurrency, and synchronization.

Predecessor

The predecessor is a comma-separated list of sequence numbers followed by a slash (‘/’):

sequence-number ‘,’ . . . ‘/’

The clause is omitted if the list is empty.

Each sequence-number is a sequence-expression without any recurrence terms. It must match the sequence number of another message.

The meaning is that the message flow is not enabled until all of the message flows whose sequence numbers are listed have occurred (a thread can go beyond the required message flow and the guard remains satisfied). Therefore, the guard condition represents a synchronization of threads.

Note that the message corresponding to the numerically preceding sequence number is an implicit predecessor and need not be explicitly listed. All of the sequence numbers with the same prefix form a sequence. The numerical predecessor is the one in which the final term is one less. That is, number 3.1.4.5 is the predecessor of 3.1.4.6.

Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (‘:’). Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[*integer* | *name*] [*recurrence*]

The integer represents the sequential order of the message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows message 3.1.3 within activation 3.1.

The name represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: message 3.1a and message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more messages that are executed depending on the conditions involved. The choices are:

‘*’ ‘[’ iteration-clause ‘]’An iteration

‘[’ condition-clause ‘]’A branch

An iteration represents a sequence of messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: *[i := 1..n].

A condition represents a message whose execution is contingent on the truth of the condition clause. The condition-clause is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: [x > y].

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): *||.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

Signature

A signature is a string that indicates the name, the arguments, and the return value of an operation, message, or signal. These have the following properties.

Return-value

This is a list of names that designates the values returned by the message within the subsequent execution of the overall interaction. These identifiers can be used as arguments to subsequent messages. If the message does not return a value, then the return value and the assignment operator are omitted.

Message-name

This is the name of the event raised in the target object (which is often the event of requesting an operation to be performed). It may be implemented in various ways, *one* of which is an operation call. If it is implemented as a procedure call, then this is the name of the operation, and the operation must be defined on the class of the receiver or inherited by it. In other cases, it may be the name of an event that is raised on the receiving object. In normal practice with procedural overloading, both the message name and the argument list types are required to identify a particular operation.

Argument list

This is a comma-separated list of arguments (actual parameters) enclosed in parentheses. The parentheses can be used even if the list is empty. Each argument is an expression in pseudocode or an appropriate programming language (UML does not prescribe). The expressions may use return values of previous messages (in the same scope) and navigation expressions starting from the source object (that is, attributes of it and links from it and paths reachable from them).

3.65.3 Presentation Options

Instead of text expressions for arguments and return values, data tokens may be shown near a message. A token is a small circle labeled with the argument expression or return value name. It has a small arrow on it that points along the message (for an argument) or opposite the message (for a return value). Tokens represent arguments and return values. The choice of text syntax or tokens is a presentation option.

The syntax of messages may instead be expressed in the syntax of a programming language, such as C++ or Smalltalk. All of the expressions on a single diagram should use the same syntax, however.

3.65.4 Example

See Figure 3-37 on page 3-92 for examples within a diagram.

Samples of control message label syntax:

2: display (x, y)	simple message
1.3.1: p:= find(specs)	nested call with return value
[x < 0] 4: invert (x, color)	conditional message
A3,B4/ C3.1*: update ()	synchronization with other threads, iteration

3.65.5 Mapping

A message flow symbol maps into a Message between the ClassifierRoles corresponding to the boxes connected by the association path bearing the message flow symbol. The control flow type sets the corresponding Message properties.

The predecessor expression, together with the sequence expression, determines the *predecessor* and *activation* (caller) associations between the Message and other messages. The predecessors of the Message are the messages corresponding to the sequence numbers in the predecessor list as well as the message corresponding to the immediate preceding sequence number as the Message (i.e., 1.2.2 is the one preceding 1.2.3). The caller of the Message is the Message whose sequence number is truncated by one position (i.e., 1.2 is the caller of 1.2.3).

The return value maps into a message from the called object to the caller with direction *return*. Its *predecessor* is the final message within the procedure. Its *activation* is the message that called the procedure.

The recurrence expression, the iteration clause, and the condition clause determine the recurrence value in the Message.

The operation name and the form of the signature determine the Operation attached to the CallAction associated with the Message.

The arguments of the signature determine the arguments associated with the CallAction.

In a procedural interaction, each message flow symbol also maps into a second Message with the properties (synchronous, reply) representing the return flow. This Message has an *activation* Association to the original call Message. Its associated Action is a ReturnAction bearing the return values as arguments (if any).

3.66 Creation/Destruction Markers

3.66.1 Semantics

During the execution of an interaction some objects and links are created and some are destroyed. The creation and destruction of elements can be marked.

3.66.2 Notation

An object or link that is created during an interaction has the keyword *new* as a constraint. An object or link that is destroyed during an interaction has the keyword *destroyed* as a constraint. The keyword may be used even if the element has no name. Both keywords may be used together, but the keyword *transient* may be used in place of *new destroyed*.

3.66.3 Presentation options

Tools may use other graphic markers in addition to or in place of the keywords. For example, each kind of lifetime might be shown in a different color. A tool may also use animation to show the creation and destruction of elements and the state of the system at various times.

3.66.4 Example

See Figure 3-37 on page 3-92.

3.66.5 Mapping

Creation or destruction indicators map into CreateActions or DestroyActions actions on the target ClassifierRoles or into TerminateActions. The actions correspond to messages that cause the changes. These status indicators are merely summaries of the total actions.

Part 9 - Statechart Diagrams

A statechart diagram shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

The semantics and notation described in this chapter are substantially those of David Harel's statecharts with modifications to make them object-oriented. His work was a major advance on the traditional flat state machines. Statechart notation also implements aspects of both Moore machines and Mealy machines, traditional state machine models.

3.67 Statechart Diagram

3.67.1 Semantics

A state machine is a graph of states and transitions that describes the response of an object of a given class to the receipt of outside stimuli. A state machine is attached to a class or a method.

3.67.2 Notation

A statechart diagram represents a state machine. The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols. States may also contain subdiagrams by physical containment and tiling.

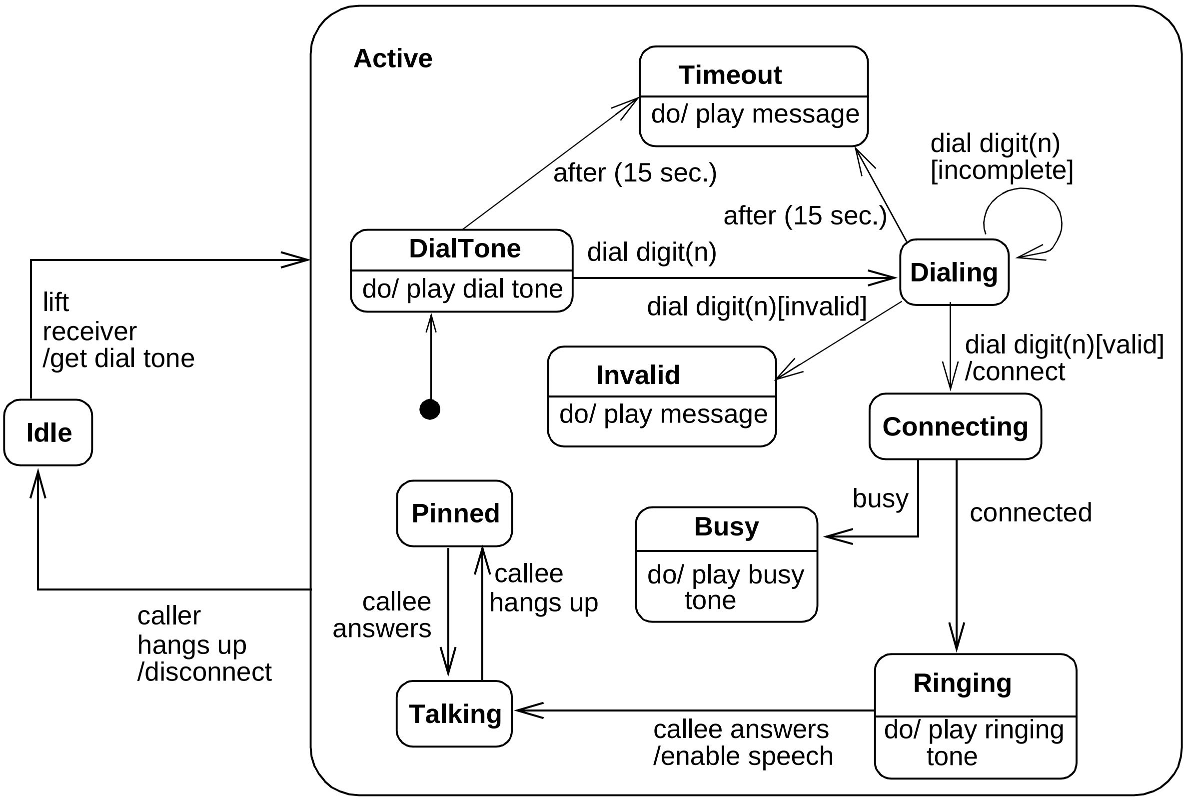


Figure 3-41 State Diagram

3.67.3 Mapping

A statechart diagram maps into a StateMachine. That StateMachine may be attached to a Class or a Method, but there is no explicit notation for this.

3.68 States

3.68.1 Semantics

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. An object remains in a state for a finite (non-instantaneous) time.

Actions are atomic and non-interruptible. A state may correspond to ongoing activity. Such activity is expressed as a nested state machine. Alternately, ongoing activity may be represented by a pair of actions, one that starts the activity on entry to the state and one that terminates the activity on exit from the state.

Each subregion of a state may have initial states and final states. A transition to the enclosing state represents a transition to the initial state. A transition to a final state represents the completion of activity in the enclosing region. Completion of activity in all concurrent regions represents completion of activity by the enclosing state and triggers a “completion of activity” event” on the enclosing state. Completion of the outermost state of an object corresponds to its death.

3.68.2 Notation

A state is shown as a rectangle with rounded corners. It may have one or more compartments. The compartments are all optional. They are as follows:

- Name compartment

  Holds the (optional) name of the state as a string. States without names are “anonymous” and are all distinct It is undesirable to show the same named state twice in the same diagram, as confusion may ensue.

- Internal transition compartment

  Holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing state. These have the format:

  event-name argument-list ‘[’ guard-condition ‘]’‘/’ *action-expression*

Each event name or pseudo-event name may appear may appear more than once per state if the guard conditions are different. The following special actions have the same form, but represent reserved words that cannot be used for event names:

‘entry’ ‘/’ *action-expression*

An atomic action performed on entry to the state

‘exit’ ‘/’ *action-expression*

An atomic action performed on exit from the state

Entry and exit actions may not have arguments or guard conditions (because they are invoked implicitly, not explicitly). However, the entry action at the top level of the state machine for a class may have parameters that represent the arguments that it receives when it is created.

Action expressions may use attributes and links of the owning object and parameters of incoming transitions (if they appear on all incoming transitions).

The following keyword represents the invocation of a nested state machine:

‘do’ ‘/’ *machine-name* (*argument-list*)

The *machine-name* must be the name of a state machine that has an initial and final state. If the nested machine has parameters, then the argument list must match correctly. When this state is entered after any entry action, then execution of the nested state machine begins with its initial state. When the nested state machine reaches its final state, any exit action in the current state is performed. The current state is considered completed and may take a transition based on implicit completion of activity.

3.68.3 Example

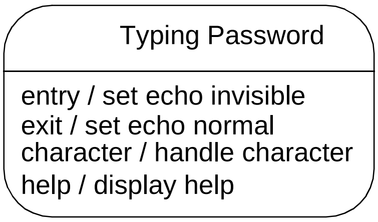


Figure 3-42 State

3.68.4 Mapping

A state symbol maps into a State. See “Composite States” on page 3-109 for further details on which kind of state.

The name string in the symbol maps to the name of the state. Two symbols with the same name map into the same state. However, each state symbol with no name (or an empty name string) maps into a distinct anonymous State.

- An internal action string with the name “entry” or “exit” maps into an association.
  - The source is the State corresponding to the enclosing state symbol.
  - The target is an ActionSequence that maps the action expression.
  - The association is the Entry action or the Exit action association.
- An internal action string with the name “do” maps into the invocation of a nested state machine.

Any other internal action maps into an internalTransition from the corresponding State to a Transition. The action expression maps into the ActionSequence and Guard for the Transition. The event name and arguments map into an Event corresponding to the event name and arguments. The Transition has a *trigger* Association to the Event.

3.69 Composite States

3.69.1 Semantics

A state can be decomposed using *and*-relationships into concurrent substates or using *or*-relationships into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Its substates may be refined in the same way or the other way.

A newly-created object starts in its initial state. The event that creates the object may be used to trigger a transition from the initial state symbol. An object that transitions to its outermost final state ceases to exist.

3.69.2 Notation

An expansion of a state shows its fine structure. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a region holding a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

An expansion of a state into concurrent substates is shown by tiling the graphic region of the state using dashed lines to divide it into subregions. Each subregion is a concurrent substate. Each subregion may have an optional name and must contain a nested state diagram with disjoint states. The text compartments of the entire state are separated from the concurrent substates by a solid line.

An expansion of a state into disjoint substates is shown by showing a nested state diagram within the graphic region.

An initial (pseudo) state is shown as a small solid filled circle. In a top-level state machine, the transition from an initial state may be labeled with the event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition to the enclosing state. The initial transition may have an action. The initial state is a notational device. An object may not be *in* such a state, but must transition to an actual state.

A final (pseudo) state is shown as a circle surrounding a small solid filled circle (a bull's eye). It represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event (usually displayed as an unlabeled transition).

3.69.3 Example

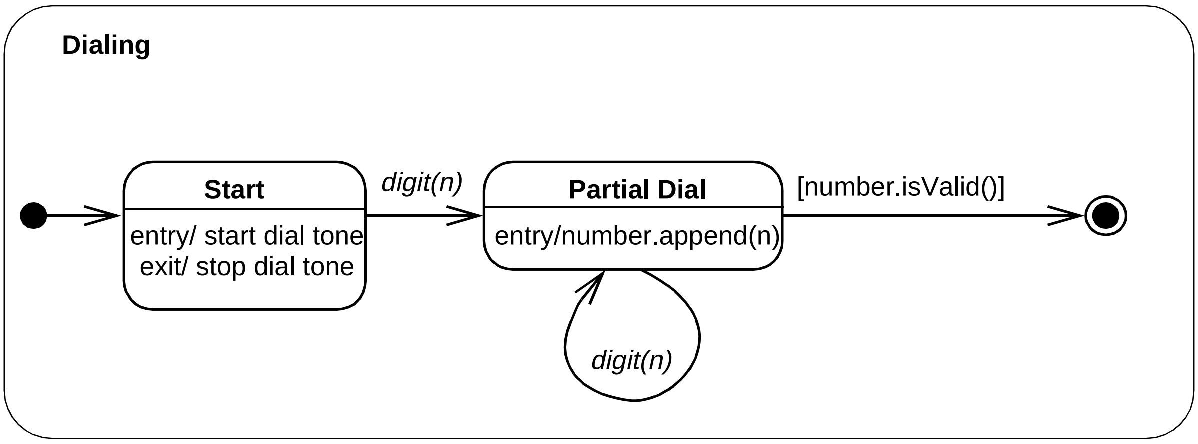


Figure 3-43 Sequential Substates

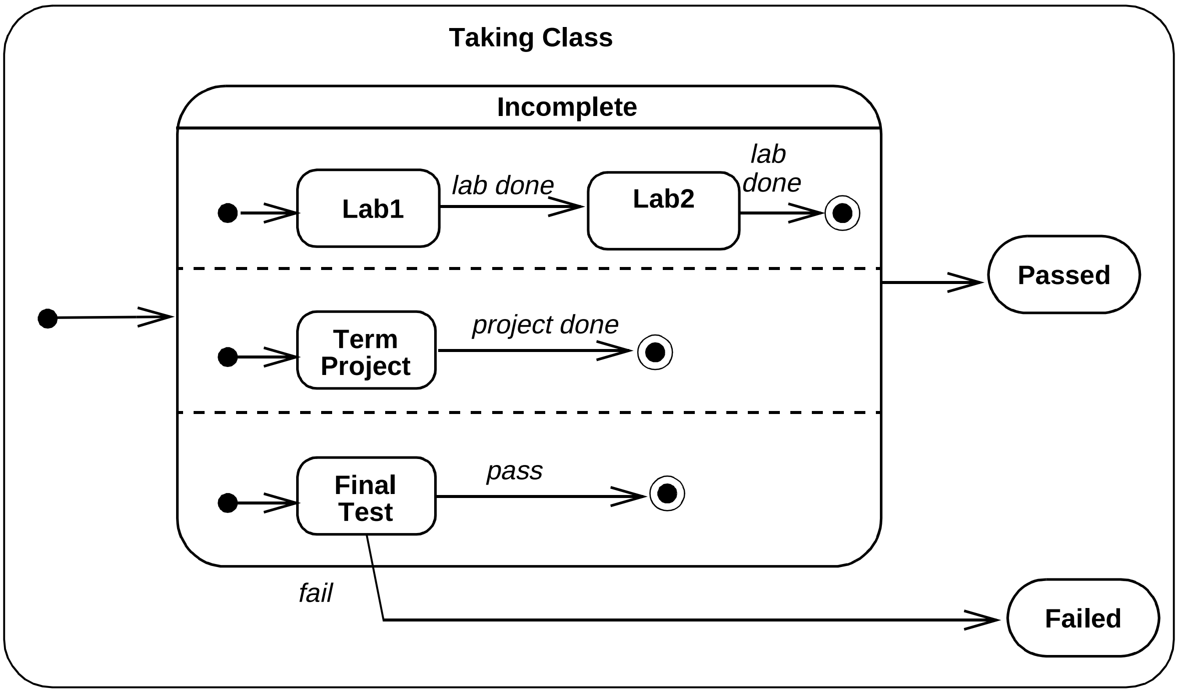


Figure 3-44 Concurrent Substates

3.69.4 Mapping

A state symbol maps into a State. If the symbol has no subdiagrams in it, it maps into a SimpleState. If it is tiled by dashed lines into subregions, then it maps into a CompositeState with the *isConcurrent* value true; otherwise, it maps into a CompositeState with the *isConcurrent* value false.

An initial state symbol or a final state symbol map into a Pseudostate of kind *initial* or *final.*

3.70 Events

3.70.1 Semantics

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily mutually exclusive).

- A designated condition becoming true (usually described as a boolean expression) is a ChangeEvent. These are notated with the keyword **when** followed by a boolean expression in parentheses. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated *once* whenever its event fires. If it is false, then the transition does not occur and the event is lost. Example: **when** (balance < 0).

- Receipt of an explicit signal from one object to another is a SignalEvent. One of these is notated by the signature of the event as a trigger on a transition.
- Receipt of a call for an operation by an object is a CallEvent. These are notated by the signature of the operation as a trigger on a transition. There is no visual difference from a signal event, it is assumed that the names distinguish them.
- Passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is a TimeEvent. These are notated as time expressions as triggers on transitions. One common time expression is the passage of time since the entry to the current state. This is notated with the keyword **after** followed by an amount of time in parentheses. Example: **after** (10 seconds).

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. An event is *not* local to a single class.

3.70.2 Notation

A signal or call event can be defined using the following format:

event-name ‘(‘ *comma-separated-parameter-list* ‘)’

A parameter has the format:

parameter-name ‘:’ *type-expression*

A signal can be declared using the «signal» keyword on a class symbol in a class diagram. The parameters are specified as attributes. A signal can be specified as a subclass of another signal. This indicates that an occurrence of the subevent triggers any transition that depends on the event or any of its ancestors.

An elapsed-time event can be specified with the keyword **after** followed by an expression that evaluates (at modeling time) to an amount of time, such as “**after** (5 seconds)” or **after** (10 seconds since exit from state A).” If no starting point is indicated, then it is the time since the entry to the current state. Other time events can be specified as conditions, such as **when** (date = Jan. 1, 2000).

A condition becoming true is shown with the keyword **when** followed by a boolean expression. This may be regarded as a continuous test for the condition until it is true, although in practice it would only be checked on a change of values (and there are ways to determine when it must be checked). This is mapped into a ChangeEvent in the model.

Signals can be declared on a class diagram with the keyword «signal» on a rectangle symbol. These define signal names that may be used to trigger transitions. Their parameters are shown in the attribute compartment. They have no operations. They may appear in a generalization hierarchy. Note that they are *not* real classes and may not appear in relationships to real classes.

3.70.3 Example

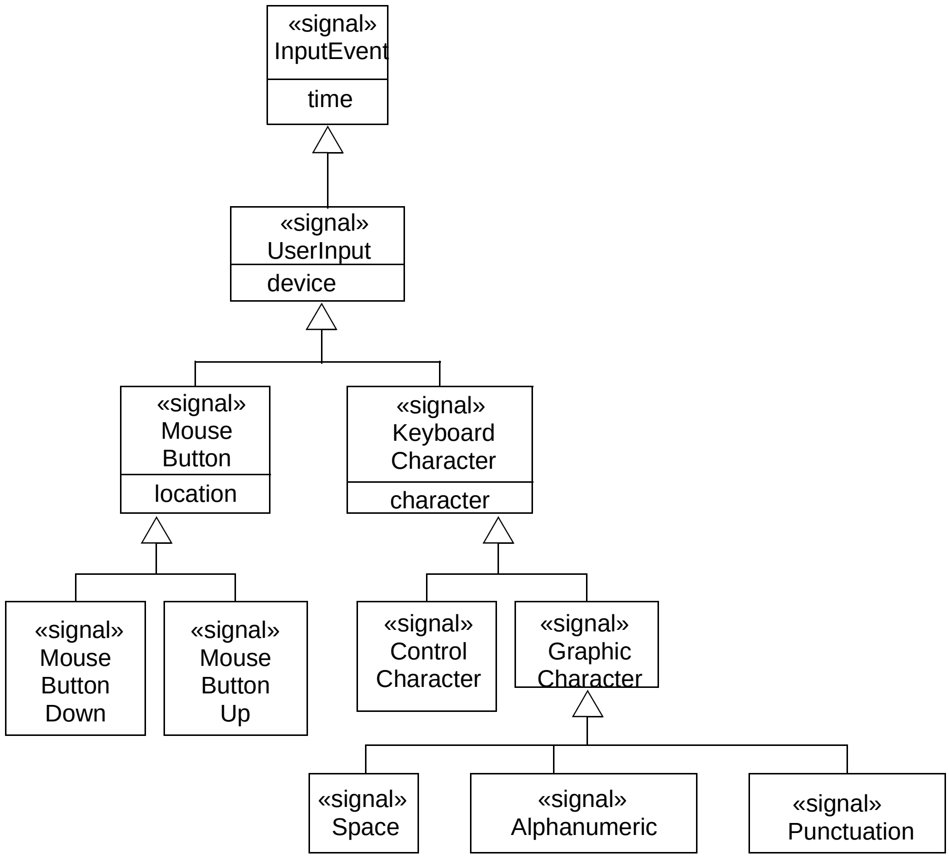


Figure 3-45 Signal Declaration

3.70.4 Mapping

A class box with stereotype «signal» maps into a Signal. The name and parameters are given by the name string and the attribute list of the box. Generalization arrows between signal class boxes map into Generalization relationships between the Signal.

The usage of an event string expression in a context requiring an event maps into an implicit reference of the Event with the given name. It is an error if various uses of the same name (including any explicit declarations) do not match.

3.71 Simple Transitions

3.71.1 Semantics

A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain specified actions when a specified event occurs, if specified conditions are satisfied. On such a change of state, the transition is said to “fire.” The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are available within actions specified on the transition or within actions initiated in the subsequent state. Events are processed one at a time. If an event does not trigger any transition, it is simply ignored. If it triggers more than one transition within the same sequential region (i.e., not in different concurrent regions), only one will fire. The choice may be nondeterministic if a firing priority is not specified.

3.71.2 Notation

A transition is shown as a solid arrow from one state (the *source* state) to another state (the *target* state) labeled by a *transition string.* The string has the following format:

event-signature ‘[’ guard-condition ‘]’ ‘/’ action-expression ‘^’ send-clause

The *event-signature* describes an event with its arguments:

event-name ‘(’ *parameter* ‘,’ . . . ‘)’

The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable object (for example, “**in** State1” or “**not in** State2”). State names may be fully qualified by the nested states that contain them, yielding path names of the form “State1::State2::State3.” This may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is a procedural expression that is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the triggering event. The action-clause must be an atomic operation, that is, it may not be interruptible. It must be executed entirely before any other actions are considered. The transition may contain more than one action clause (with delimiter).

‘The *send-clause* is a special case of an action, with the format:

destination-expression ‘.’ *destination-message-name* ‘(‘ *argument* ‘.’ . . . ‘)’

The transition may contain more than one send clause (with delimiter). The relative order of action clauses and send clauses is significant and determines their execution order.

The *destination-expression* is an expression that evaluates to an object or a set of objects.

The *destination-message-name* is the name of a message (operation or signal) meaningful to the destination object(s).

The *destination-expression* and the *arguments* may be written in terms of the parameters of the triggering event and the attributes and links of the owning object.

Branches

A simple transition may be extended to include a tree of decision symbols (see “Decisions” on page 3-127). This is equivalent to a set of individual transitions, one for each path through the tree, whose guard condition is the “and” of all of the conditions along the path.

Transition times

Names may be placed on transitions to designate the times at which they fire. See “Transition Times” on page 3-89.

3.71.3 Example

right-mouse-down (location) [location in window] / object := pick-object (location) ^ object.highlight ()

The event may be any of the types. Selecting the type depends on the syntax of the name (for time events, for example); however, SignalEvents and CallEvents are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

3.71.4 Mapping

A transition string and the transition arrow that it labels together map into a Transition and its attachments. The arrow connects two state symbols. The Transition has the corresponding States as its source (the state at the tail) and destination (the state at the head) States in associations to the Transition.

The event name and parameters map into an Event element, which may be a SignalEvent, a CallEvent, or a TimeExpression (if it has the proper syntax). The event is attached as a *trigger* Association to the Transition.

The guard condition maps into a Guard element attached to the Transition.

An action expression maps into an ActionSequence attached as an *effect* Association to the Transition. The target object expression (if any) in the expression maps into a *target* ObjectSetExpression. Each term in the action expression maps into an Action that is a part of the ActionSequence. A send clause maps into a RaiseAction with an ObjectSetExpression for the destination.

A transition time label on a transition maps into a TimingMark attached to the Transition.

3.72 Complex Transitions

A complex transition may have multiple source states and target states. It represents a synchronization and/or a splitting of control into concurrent threads without concurrent substates.

3.72.1 Semantics

A complex transition is enabled when all the source states are occupied. After a complex transition fires, all its destination states are occupied.

3.72.2 Notation

A complex transition is shown as a short heavy bar (a *synchronization* bar, which can represent synchronization, forking, or both). The bar may have one or more solid arrows from states to the bar (these are the *source states*). The bar may have one or more solid arrows from the bar to states (these are the *destination states*). A transition string may be shown near the bar. Individual arrows do not have their own transition strings.

3.72.3 Example

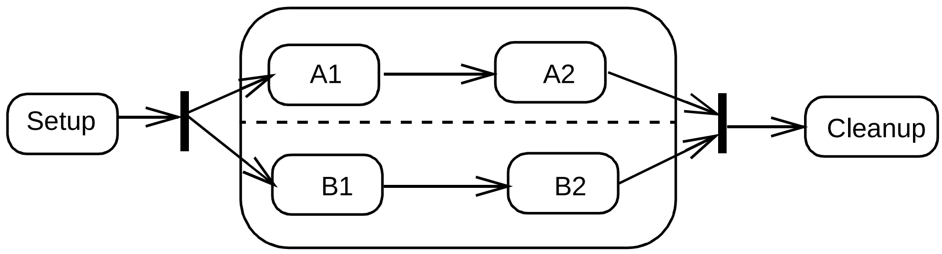


Figure 3-46 Complex Transition

3.72.4 Mapping

A bar with multiple transition arrows leaving it maps into a fork Pseudostate. A bar with multiple transition arrows entering it maps into a join Pseudostate. The Transitions corresponding to the incoming and outgoing arrows attach to the pseudostate as if it were a regular state. If a bar has multiple incoming and multiple outgoing arrows, then it maps into a Join connected to a Fork pseudostate by a single Transition with no attachments.

3.73 Transitions to Nested States

3.73.1 Semantics

A transition drawn to the boundary of a complex state is equivalent to a transition to its initial state (or to a complex transition to the initial states of each of its concurrent subregions, if it is concurrent). The entry action is always performed when a state is entered from outside.

A transition from a complex state indicates a transition that applies to each of the states within the state region (at any depth). It is "inherited" by the nested states. Inherited transitions can be masked by the presence of nested transitions with the same trigger.

3.73.2 Notation

A transition drawn to a complex state boundary indicates a transition to the complex state. This is equivalent to a transition to the initial state within the complex state region. The initial state must be present. If the state is a concurrent complex state, then the transition indicates a transition to the initial state of each of its concurrent substates.

Transitions may be drawn directly to states within a complex state region at any nesting depth. All entry actions are performed for any states that are entered on any transition. On a transition within a concurrent complex state, transition arrows from the synchronization bar may be drawn to one or more concurrent states. Any other concurrent subregions start with their default initial states.

A transition drawn from a complex state boundary indicates a transition of the complex state. If such a transition fires, any nested states are forcibly terminated and perform their exit actions, then the transition actions occur and the new state is established.

Transitions may be drawn directly from states within a complex state region at any nesting depth to outside states. All exit actions are performed for any states that are exited on any transition. On a transition from within a concurrent complex state, transition arrows may be specified from one or more concurrent states to a synchronization bar; therefore, specific states in the other regions are irrelevant to triggering the transition.

A state region may contain a *history state indicator* shown as a small circle containing an 'H.' The history indicator applies to the state region that directly contains it. A history indicator may have any number of incoming transitions from outside states. It may have at most one outgoing unlabeled transition. This identifies the default "previous state" if the region has never been entered. If a transition to the history indicator fires, it indicates that the object resumes the state it last had within the complex region. Any necessary entry actions are performed. The history indicator may also be 'H*' for *deep history.* This indicates that the object resumes the state it last had at any depth within the complex region, rather than being restricted to the state at the same level as the history indicator. A region may have both shallow and deep history indicators.

3.73.3 Presentation options

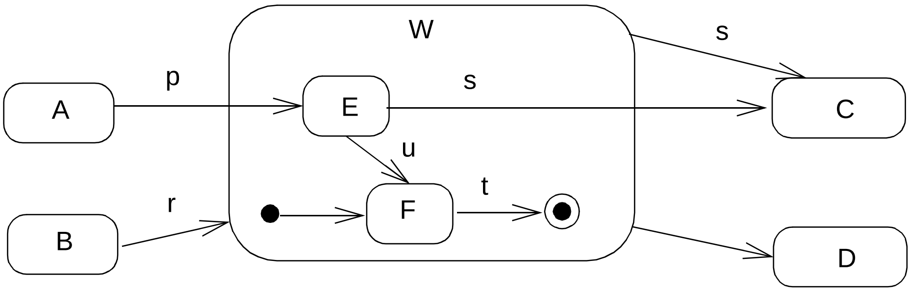
Stubbed transitions

Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial state may (but need not) be shown as coming from or going to *stubs*. A *stub* is shown as a small vertical line drawn inside the boundary of the enclosing state. It indicates a transition connected to a suppressed internal state. Stubs are not used for transitions to initial or from final states.

Note that events should be shown on transitions leading into a state, either to the state contour or to an internal substate, including a transition to a stubbed state. Normally events should not be shown on transitions leading from a stubbed state to an external state. Think of a transition as belonging to its source state. If the source state is suppressed, then so are the details of the transition. Note also that a transition from a final state is summarized by an unlabeled transition from the complex state contour (denoting the implicit event “action complete” for the corresponding state).

3.73.4 Example

See Figure 3-44 on page 3-111 and Figure 3-46 on page 3-116 for examples of complex transitions. Following are examples of stubbed transitions and the history indicator.



may be abstracted as

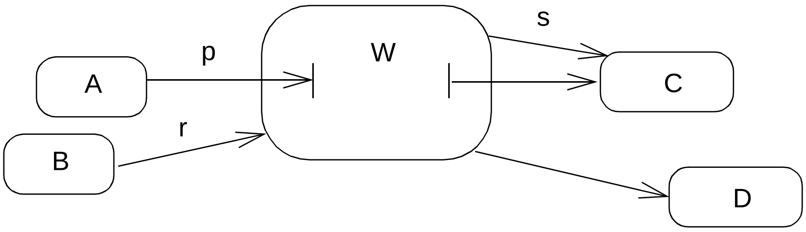


Figure 3-47 Stubbed Transitions

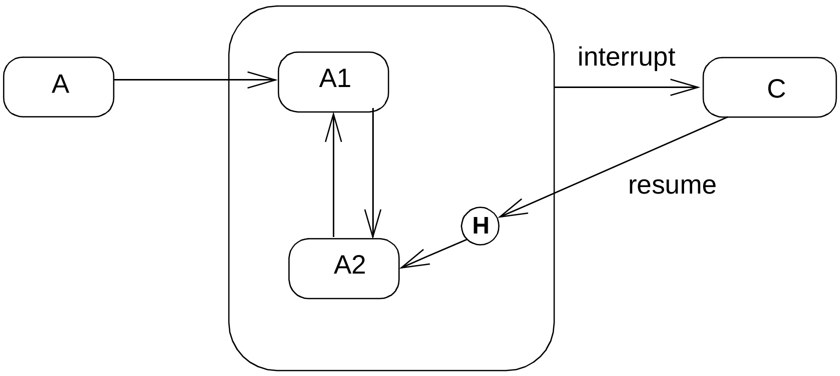


Figure 3-48 History Indicator

3.73.5 Mapping

An arrow to any state boundary, nested or not, maps into a Transition between the corresponding States and similarly for transitions directly to history states.

A history indicator maps into a Pseudostate of kind *shallowHistory* or *deepHistory.*

A stubbed transition does not map into anything in the model. It is a notational elision that indicates the presence of transitions to additional states in the model that are not visible in the diagram.

3.74 Sending Messages

3.74.1 Semantics

Messages are sent by an action in an object to a target set of objects. The target set can be a single object, the entire system, or some other set. The sender can be subsumed to an object, a composite object, or a class.

3.74.2 Notation

See "Location of Components and Objects within Objects" on page 3-141 for the text syntax of sending messages that cause events for other objects.

Sending such a message can also be shown visually. See "Object Lifeline" on page 3-86 and "Message flows" on page 3-101 for details of showing messages in sequence diagrams and collaboration diagrams.

Sending a message between state diagrams may be shown by drawing a dashed arrow from the sender to the receiver. Messages must be sent between objects, so this means that the diagram must be some form of object diagram containing objects (not classes). The arrow is labeled with the event name and arguments of the event that is caused by the reception of the event. Each state diagram must be contained within an object symbol representing a collaborating object. Graphically, the state diagrams may be nested physically within an object symbol, or the object enclosing *one* state diagram may be implicit (being the object owning the main state diagram at issue). The state diagrams represent the states of the collaborating objects.

Note that this notation may also be used on other kinds of diagrams to show sending of events between classes or objects.

The sender symbol may be one of:

- A transition. The message is sent as part of the action of firing the transition. This is an alternate presentation to the text syntax for sending messages.
- An object. The message is sent by an object of the class at some point in its life, but the details are unspecified.

The receiver may be one of:

- An object, including a class reference symbol containing a state diagram. The message is received by the object and may trigger a transition on the corresponding event. There may be many transitions involving the event. This notation may not be used when the target object is computed dynamically. In that case, a text expression must be used.

- A transition. The transition must be the only transition in the object involving the given event, or at least the only transition that could possibly be triggered by the particular sending of the message. This notation may not be used when the transition triggered depends on the state of the receiving object and not just on the sender.
- A class designation. This notation would be used to model the invocation of class-scope operations, such as the creation of a new instance. The receipt of such a message causes the instantiation of a new object in its default initial state. The event seen by the receiver may be used to trigger a transition from its default initial state and represents a way to pass information from the creator to the new object.

3.74.3 Example

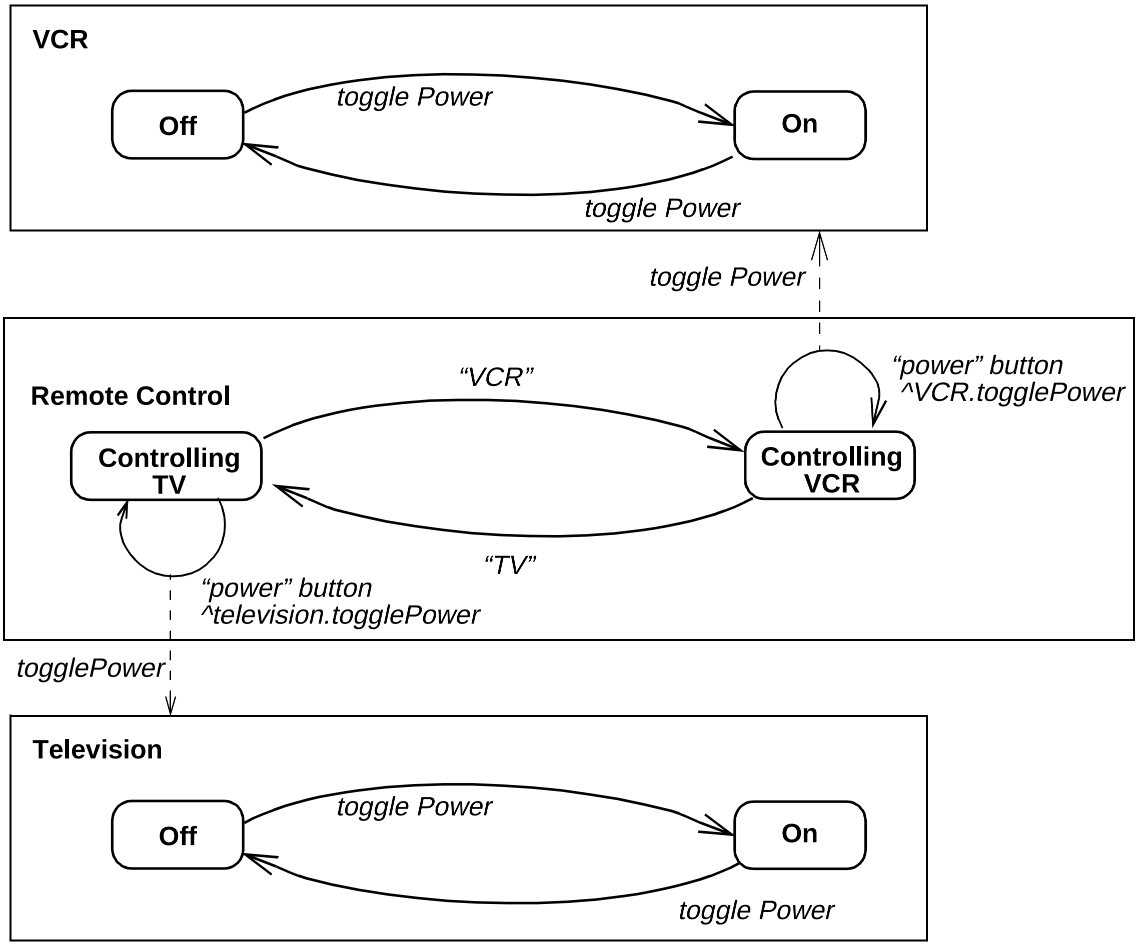


Figure 3-49 Sending Messages

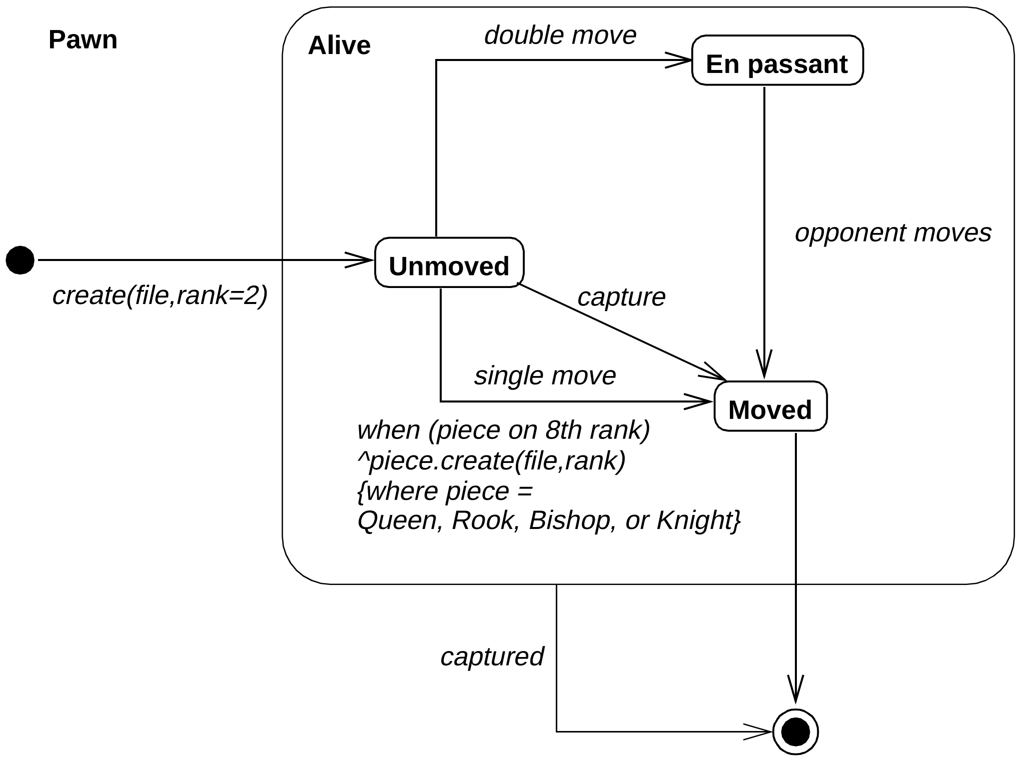


Figure 3-50 Creating and Destroying Objects

3.74.4 Mapping

A send arrow to an object maps into a SendAction whose *message* is a Signal that corresponds to the name on the arrow and whose *target* ObjectSetExpression corresponds to the target object.

If the arrow goes directly to a transition in the target object statechart, then the *target* ObjectSetExression corresponds to the object owning the statechart containing the transition. In addition, the transition in the target statechart implicitly triggers on the event being sent (i.e., the name of the sent event is effectively written on the target transition).

If the sender symbol is an object, then the diagram is suggestive of the sender but has no actual semantic mapping.

3.75 Internal Transitions

3.75.1 Semantics

An internal transition is a transition that remains within a single state rather than a transition that involves two states. It represents the occurrence of an event that does not cause a change of state. Entering the state (from any other state not nested in the particular state) and exiting the state (to any other state not nested in the particular state) are treated notationally as internal transitions with the reserved words “entry” and “exit;” however, they are not really internal transitions in the internal model.

Note that an internal transition is not equivalent to a self-transition from a state back to the same state. The self-transition causes the exit and entry actions on the state to be executed and the initial state to be entered, whereas the internal transition does not invoke the exit and entry actions and does not cause a change of state (including a nested state).

3.75.2 Notation

An internal transition is attached to the state rather than a transition. Graphically it is shown as a text string within the internal transition compartment on a state symbol. The syntax of an internal transition string is the same as for an external transition. See “Simple Transitions” on page 3-114 for details.

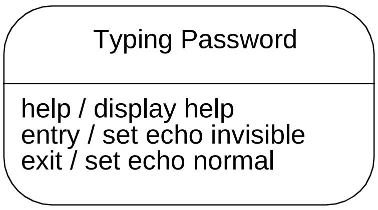


Figure 3-51 State with Internal Transitions

3.75.3 Mapping

The mapping for internal transitions has been given in “Mapping” on page 3-109.

Part 10 - Activity Diagrams

3.76 Activity Diagram

3.76.1 Semantics

An activity model is a variation of a state machine in which the states are Activities representing the performance of operations and the transitions are triggered by the completion of the operations. It represents a state machine of a procedure itself, the procedure is the implementation of an operation on the owning class.

3.76.2 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action states and in which all (or at least most) of the transitions are triggered by completion of the actions in the source states. The entire activity diagram is attached (through the model) to a class or to the implementation of an operation or a use case. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). Use ordinary state diagrams in situations where asynchronous events occur.

3.76.3 Example

Person::Prepare Beverage

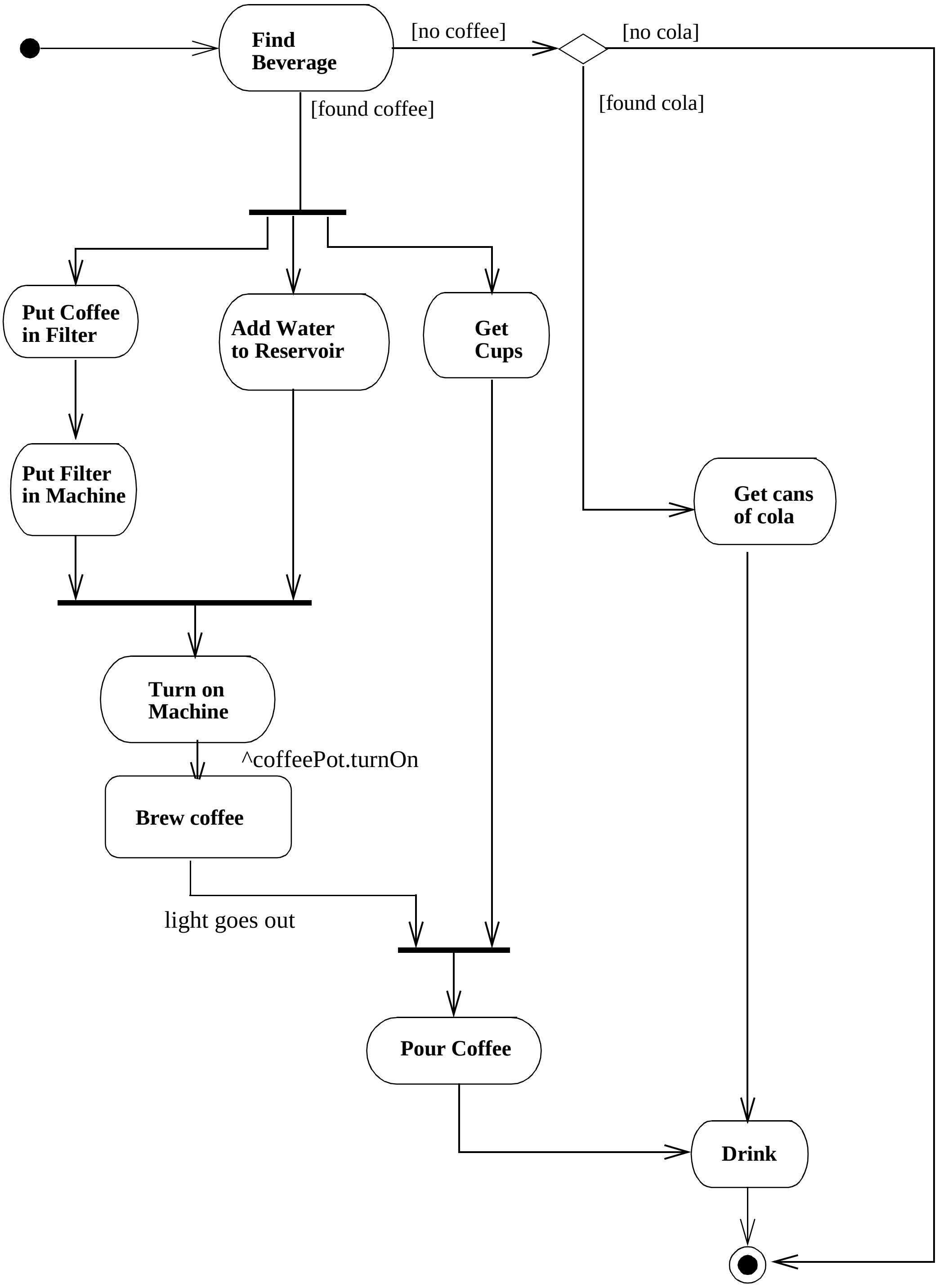


Figure 3-52 Activity Diagram

3.76.4 Mapping

An activity diagram maps into an ActivityModel.

3.77 Action state

3.77.1 Semantics

An *action state* is a shorthand for a state with an internal action and at least one outgoing transition involving the implicit event of completing the internal action (there may be several such transitions if they have guard conditions). Action states should not have internal transitions or outgoing transitions based on explicit events, use normal states for this situation. The normal use of an action state is to model a step in the execution of an algorithm (a procedure).

3.77.2 Notation

An action state is shown as a shape with straight top and bottom and with convex arcs on the two sides. The *action-expression* is placed in the symbol. The action expression need not be unique within the diagram.

Transitions leaving an action state should not include an event signature. Such transitions are implicitly triggered by the completion of the action in the state. The transitions may include guard conditions and actions.

3.77.3 Presentation options

The action may be described by natural language, pseudocode, or programming language code. It may use only attributes and links of the owning object.

Note that action state notation may be used within ordinary state diagrams; however, they are more commonly used with activity diagrams, which are special cases of state diagrams.

3.77.4 Example

Figure 3-53 Activities

3.77.5 Mapping

An action state symbol maps into an ActionState invoking a CallAction. This is equivalent to an *entry* action on a regular state. There is no *exit* nor any internal transitions. The State is normally anonymous.

3.78 Decisions

3.78.1 Semantics

A state diagram (and by derivation an activity diagram) expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides shorthand for showing decisions.

3.78.2 Notation

A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one or more incoming arrows and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger. All possible outcomes should appear on one of the outgoing transitions.

Note that a chain of decisions may be part of a complex transition, but only the first segment in such a chain may contain an event trigger label. All segments may have guard expressions.

3.78.3 Example

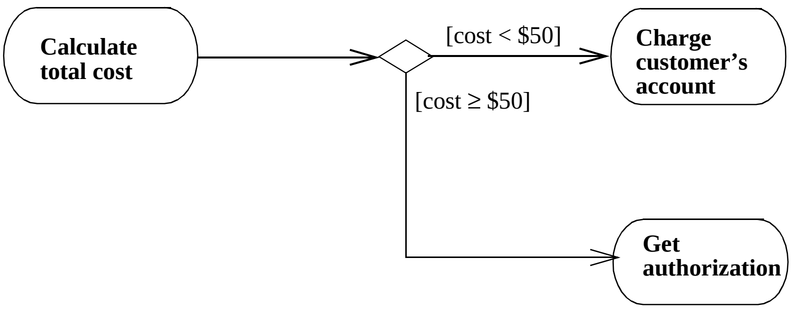


Figure 3-54 Decision

3.78.4 Mapping

A decision symbol maps into a Pseudostate of kind *branch*. Each label on an outgoing arrow maps into a Guard on the corresponding Transition, leaving the Pseudostate.

3.79 Swimlanes

3.79.1 Semantics

Actions may be organized into *swimlanes*. Swimlanes are a kind of package used to organize responsibility for activities within a class. They often correspond to organizational units in a business model.

3.79.2 Notation

An activity diagram may be divided visually into “swimlanes,” each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity, and may eventually be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance, but might indicate some affinity. Each action is assigned to one swimlane. Transitions may cross lanes. There is no significance to the routing of a transition path.

3.79.3 Example

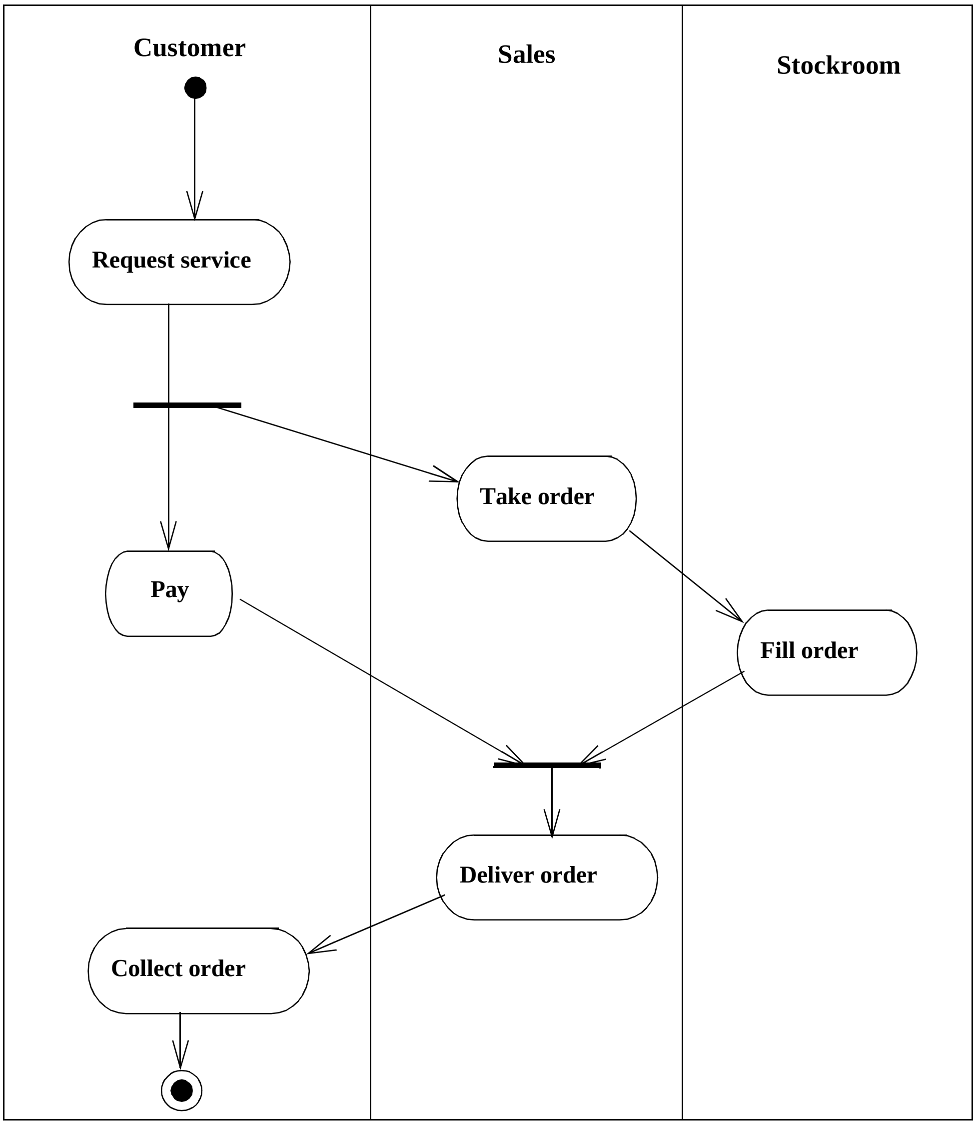


Figure 3-55 Swimlanes in Activity Diagram

3.79.4 Mapping

A swimlane maps into a Partition of the States in the ActivityModel. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

3.80 Action-Object Flow Relationships

3.80.1 Semantics

Activities operate by and on objects. Two kinds of relationships can be shown: 1) The kinds of objects that have primary responsibility for performing an action and 2) the other objects whose values are used or determined by the action. These are modeled as messages sent between the object owning the activity model and the objects that are input or output by the actions in the model.

3.80.2 Notation

Object responsible for an action

The object responsible for performing an action can be shown by drawing a lifeline and placing actions on lifelines Each lifeline represents a distinct object. There may be multiple lifelines for different objects of the same or different kinds. If this approach is chosen, usually a sequence diagram should be used. See “Sequence Diagram” on page 3-82. If an object lifeline is not shown, then some object within the swimlane package is responsible for the action, but the object is not shown. Multiple actions within a single swimlane can be handled by the same or different objects.

Object flow

Objects that are input to or output by an action may be shown as object symbols. A dashed arrow is drawn from an action outgoing transition to an output object, and a dashed arrow is drawn from an input object to an incoming arrow of an action. The same object may be (and usually is) the output of one action and the input of one or more subsequent actions.

The control flow (solid) arrows may be omitted when the object flow (dashed) arrows supply a redundant constraint. In other words, when an action produces an output that is input by a subsequent action, that object flow relationship implies a control constraint.

Object in state

Frequently the same object is manipulated by a number of successive activities. It is possible to show the arrows to and from all of the relevant activities. For greater clarity, the object may be displayed multiple times on a diagram, each appearance denoting a different point during its life. To distinguish the various appearances of the same object, the state of the object at each point may be placed in brackets and appended to the name of the object (for example, PurchaseOrder[approved]). This notation may also be used in collaboration diagrams.

3.80.3 Example

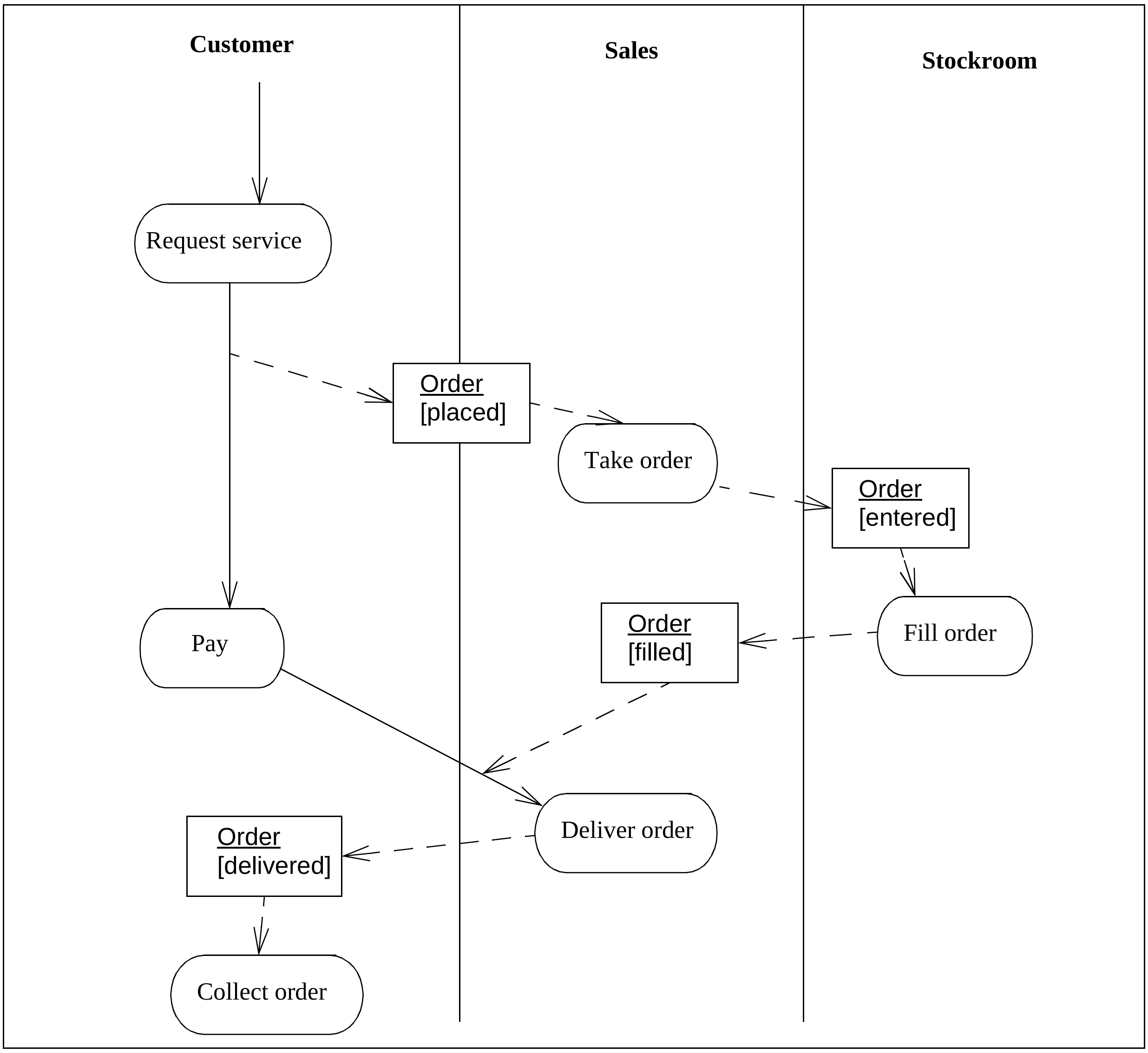


Figure 3-56 Actions and Object Flow

3.80.4 Mapping

An object flow symbol maps into an ObjectFlowState whose incoming and outgoing Transitions correspond to the incoming and outgoing arrows. The Transitions have no attachments. The class name and (optional) state name of the object flow symbol map into a Class or a ClassifierInState with the given name(s).

3.81 Control Icons

The following icons provide explicit symbols for certain kinds of information that can be specified on transitions. These icons are not necessary for constructing activity diagrams, but many users prefer the added impact that they provide.

3.81.1 Stereotypes

Signal receipt

The receipt of a signal may be shown as a concave pentagon that looks like a rectangle with a triangular notch in its side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. This symbol replaces the event label on the transition. A dashed arrow may be drawn from an object symbol to the notch on the pentagon to show the sender of the signal; this is optional.

Signal sending

The sending of a signal may be shown as a convex pentagon that looks like a rectangle with a triangular point on one side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. This symbol replaces the send-signal label on the transition. A dashed arrow may be drawn from the point on the pentagon to an object symbol to show the receiver of the signal, this is optional.

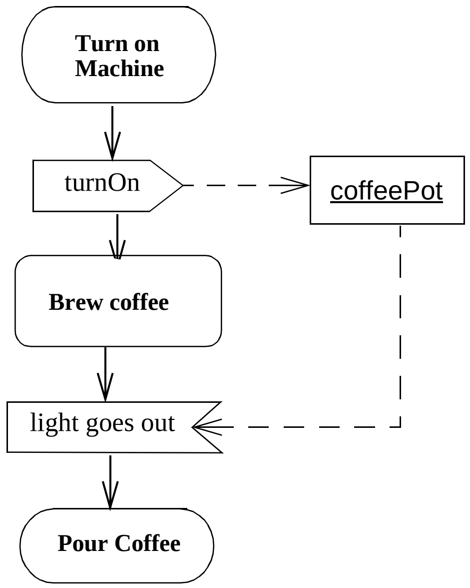


Figure 3-57 Symbols for Signal Receipt and Sending

Deferred events

A frequent situation is when an event that occurs must be “deferred” for later use while some other activity is underway. (Normally an event that is not handled immediately is lost.) This may be thought of as having an internal transition that handles the event and places it on an internal queue until it is needed or until it is discarded. Each state or activity specifies a set of events that are deferred if they occur during the state or activity. If an event is not included in the set of deferred events for a state, then it is discarded from the queue even if it has already occurred. If a transition depends on an event, the transition fires immediately if the event is already on the internal queue. If several transitions are possible, the leading event in the queue takes precedence.

A deferred event is shown by listing it within the state followed by a slash and the special operation *defer*. If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state, in which case it remains deferred throughout the composite state.

When used in conjunction with an action state, a deferred event that occurs during the action state is deferred until the action is completed, when it may trigger a transition. This means that the transition will occur correctly regardless of the relative order of the event and the action completion.

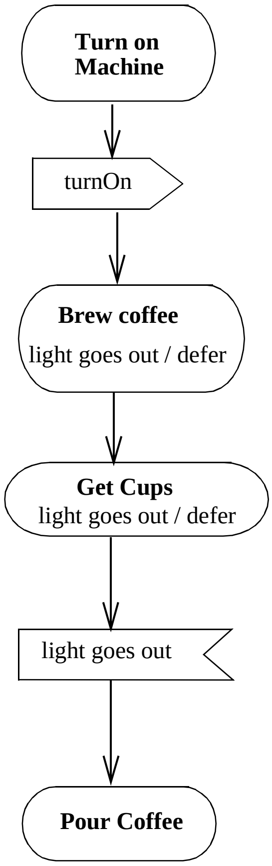


Figure 3-58 Deferred Event

3.81.2 Mapping

An input event symbol maps into an event trigger on the Transition between the States corresponding to the connected state symbols.

An output event symbol maps into a RaiseAction on the Transition between the States corresponding to the connected state symbols.

An input event symbol whose successor is a join symbol maps into an event trigger on a Transition to an implicit dummy State. The outgoing Transition from the dummy State enters the join Pseudostate.

A deferred event attached to a state maps into a *deferredEvent* association from the State to the Event.

Part 11 - Implementation Diagrams

Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms: 1) component diagrams show the structure of the code itself and 2) deployment diagrams show the structure of the run-time system.

3.82 Component Diagram

3.82.1 Semantics

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. A software module may be represented as a component type. Some components exist at compile time, some exist at link time, some exist at run time, and some exist at more than one time. A compile-only component is one that is only meaningful at compile time. The run-time component in this case would be an executable program.

A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes).

3.82.2 Notation

A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.

A diagram containing component types and node types may be used to show compiler dependencies, which are shown as dashed arrows (dependencies) from a client component to a supplier component that it depends on in some way. The kinds of dependencies are language-specific and may be shown as stereotypes of the dependencies.

The diagram may also be used to show interfaces and calling dependencies among components, using dashed arrows from components to interfaces on other components.

3.82.3 Example

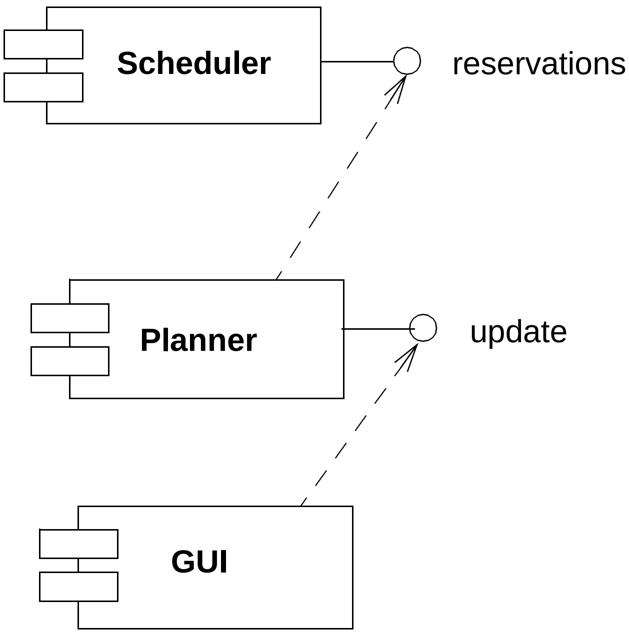


Figure 3-59 Component Diagram

3.82.4 Mapping

A component diagram maps to a static model whose elements include Components.

3.83 Deployment Diagrams

3.83.1 Semantics

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams, they should be shown on component diagrams.

3.83.2 Notation

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances. This indicates that the component lives or runs on the node. Components may contain objects, this indicates that the object is part of the component. Components are connected to other components by dashed-arrow dependencies (possibly through interfaces). This indicates that one component uses the services of another component. A stereotype may be used to indicate the precise dependency, if needed.

The deployment type diagram may also be used to show which components may run on which nodes, by using dashed arrows with the stereotype «supports».

Migration of components from node to node or objects from component to component may be shown using the «becomes» stereotype of the dependency relationship. In this case the component or object is resident on its node or component only part of the entire time.

Note that a process is just a special kind of object (see Active Object).

3.83.3 Example

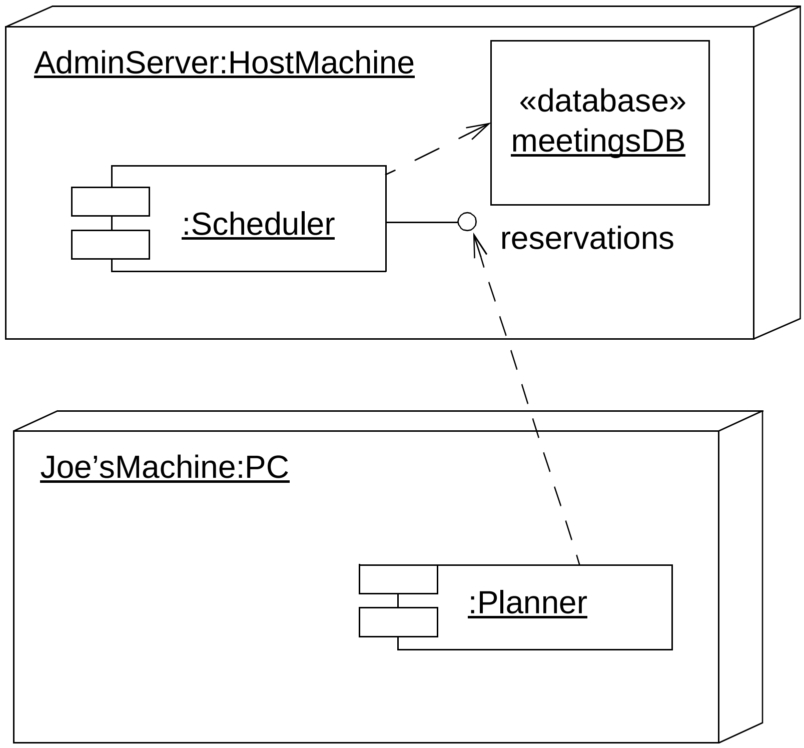


Figure 3-60 Nodes

3.83.4 Mapping

A deployment diagram maps to a static model whose elements include Nodes. It is not particularly distinguished in the model.

3.84 Nodes

3.84.1 Semantics

A node is a run-time physical object that represents a processing resource. Generally, having at least a memory and often processing capability as well. Nodes include computing devices but also human resources or mechanical processing resources. Nodes may be represented as type and as instances. Run time computational instances, both objects and component instances, may reside on node instances.

3.84.2 Notation

A node is shown as a figure that looks like a 3-dimensional view of a cube. A node type has a type name:

node-type

A node instance has a name and a type name. The node may have an underlined name string in it or below it. The name string has the syntax:

name ‘:’ *node-type*

The name is the name of the individual node (if any). The node-type says what kind of a node it is. Either or both elements are optional.

Dashed-arrow dependency arrows show the capability of a node type to support a component type. A stereotype may be used to state the precise kind of dependency.

Component instances and objects may be contained within node instance symbols. This indicates that the items reside on the node instances. Containment may also be shown by aggregation or composition association paths.

Nodes may be connected by associations to other nodes. An association between nodes indicates a communication path between the nodes. The association may have a stereotype to indicate the nature of the communication path (for example, the kind of channel or network).

3.84.3 Example

This example shows two nodes containing an object (cluster) that migrates from one node to another and an object that remains in place.

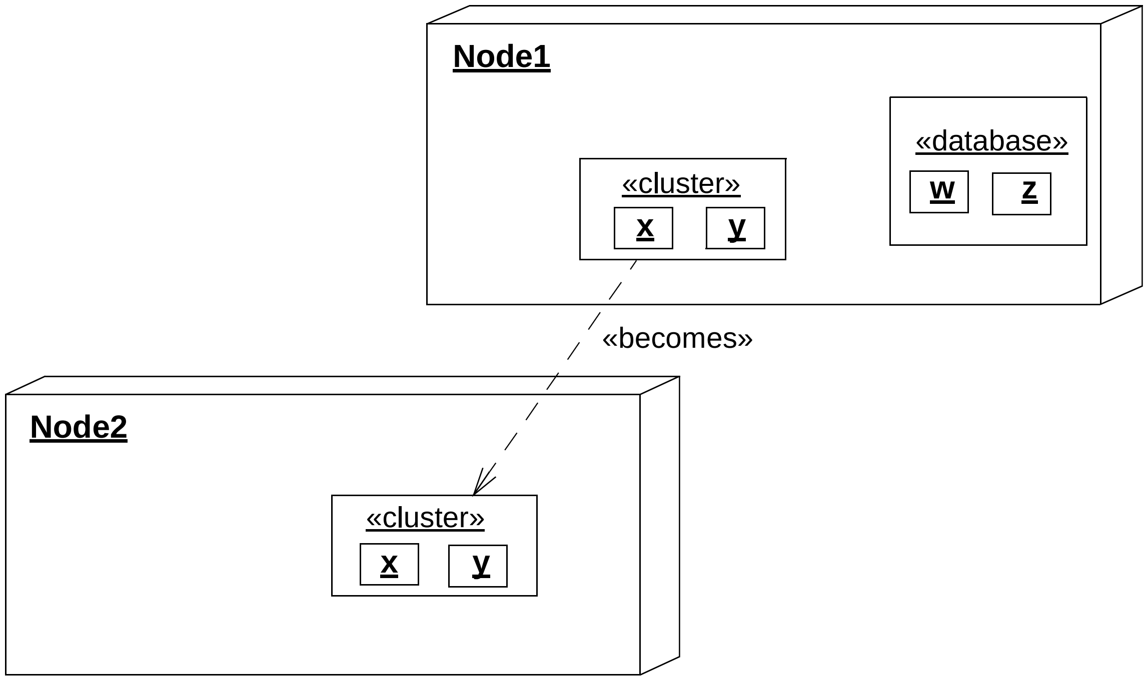


Figure 3-61 Use of Nodes to Hold Objects

3.84.4 Mapping

A node maps to a Node. The nesting of symbols within the node symbol maps into a composition association between a node and constituent Classes, or a composition link between a Node and constituent Objects.

3.85 Components

3.85.1 Semantics

A component type represents a distributable piece of implementation of a system, including software code (source, binary, or executable) but also including business documents, etc., in a human system. Components may be used to show dependencies, such as compiler and run-time dependencies or information dependencies in a human organization. A component instance represents a run-time implementation unit and may be used to show implementation units that have identity at run time, including their location on nodes.

3.85.2 Notation

A component is shown as a rectangle with two small rectangles protruding from its side. A component type has a type name:

component-type

A component instance has a name and a type. The name of the component and its type may be shown as an underlined string either within the component symbol or above or below it, with the syntax:

component-name ':' *component-type*

A property may be used to indicate the life-cycle stage that the component describes (source, binary, executable, or more than one of those). Components (including programs, DLLs, run-time linkable images, etc.) may be located on nodes.

3.85.3 Example

The example shows a component with interfaces and also a component that contains objects at run time.

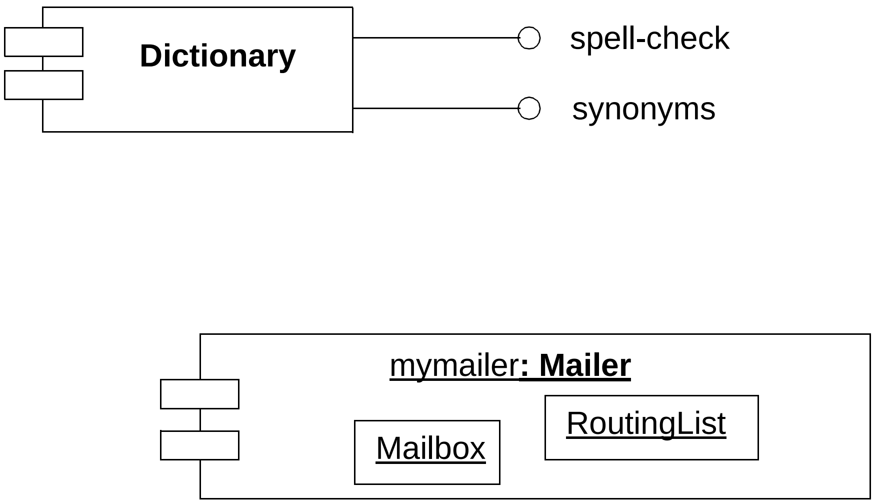


Figure 3-62 Components

3.85.4 Mapping

A component symbol maps to a Component. Graphical nesting of other symbols maps into a composition association of the Component to Classes or Objects in it.

Interface circles attached to the component symbol by solid lines map into *supports* Dependencies to Interfaces.

3.86 Location of Components and Objects within Objects

3.86.1 Semantics

Instances may be located within other instances. For example, objects may live in processes that live in components that live on nodes. In more complicated situations processes may migrate from node to node, so a process may live in many nodes and deal with many components over time.

3.86.2 Notation

The location of an instance (including objects, component instances, and node instances) within another instance may be shown by physical nesting. Containment may also be shown by aggregation or composition association paths. Alternately, an instance may have a property tag “location” whose value is the name of the containing instance.

If an object moves during an interaction, then it may be as two or more occurrences with a “becomes” dependency between the occurrences. The dependency may have a time property attached to it to show the time when the object moves. Each occurrence represents the object during a period of time. Messages should be directed to the correct occurrence of the object.

3.86.3 Example

See the other diagrams in this section for examples of objects and components located on nodes as well as migration.

3.86.4 Mapping

Physical nesting of symbols maps into composition association from the Element corresponding to the outer symbol to the Elements corresponding to the contents.

UML Extensions *4*

This chapter includes the *UML Extension for Software Development Processes* and the *UML Extension for Business Modeling*.

Contents

This chapter contains the following topics.

4

Part 1 - UML Extension for Software Development Processes

4.1 Overview

User-defined extensions of the UML are enabled through the use of stereotypes, tagged values, and constraints. Two extensions are defined currently: 1) Software Development Processes and 2) Business Modeling.

The UML is broadly applicable without extension, so companies and projects should define extensions only when they find it necessary to introduce new notation and terminology. Extensions will not be as universally understood, supported, and agreed upon as the UML itself. In order to reduce potential confusion around vendor implementation, the following terms are defined:

- UML Variant - a language with well-defined semantics that is built on top of the UML metamodel, as a metamodel. It specializes the UML metamodel, without changing any of the UML semantics or redefining any of its terms. For example, it could reintroduce a class called State.
- UML Extension - a predefined set of Stereotypes, TaggedValues, Constraints, and notation icons that collectively extend and tailor the UML for a specific domain or process.

4.2 Introduction

This section defines the UML Extension for Software Development Processes, defined in terms of the UML's extension mechanisms, namely Stereotypes, TaggedValues, and Constraints.

See the UML Semantics chapter for a full description of the UML extension mechanisms.

This chapter describes a UML extension that has been found useful in software development processes. Although this extension was based on the Objectory process, it is general-purpose and may also be applied to other software development processes. It is not meant to be a comprehensive description of all software development processes, but is an example of one such process.

4.3 Summary of Extension

Table 4-1 Stereotypes

Metamodel Class	Stereotype Name
Model	use case model
Model	analysis model
Model	design model
Model	implementation model

Table 4-1 Stereotypes

Package	use case system
Subsystem	analysis system
Subsystem	design system
Package	implementation system
Subsystem	analysis subsystem
Subsystem	design system
Package	implementation system
Package	use case package
Subsystem	analysis service package
Subsystem	design service package
Class	boundary
Class	entity
Class	control
Association	communicates
Association	subscribes
Collaboration	use case realization

4.3.1 TaggedValues

Currently, this extension does not introduce any new TaggedValues.

4.3.2 Constraints

Currently, this extension does not introduce any new Constraints, other than those associated with the well-formedness semantics of the stereotypes introduced.

4.3.3 Prerequisite Extensions

This extension requires no other extensions to the UML for its definition.

4.4 Stereotypes and Notation

4.4.1 Model, Package, and Subsystem Stereotypes

A software development process comprises several different, but related models. Software engineering process models are characterized by the lifecycle stage that they represent. The different models are stereotypes of model:

- Use Case

- Analysis
- Design
- Implementation

Use Case

A Use Case Model is a model in which the top-level package is a use case system.

A Use Case System is a top-level package. A use case system contains use case packages and/or use cases and/or actors and relationships.

A Use Case Package is a package containing use cases and actors with relationships. A use case is not partitioned over several use case packages.

Analysis

An Analysis Model is a model whose top-level package is an analysis system.

An Analysis System is a top-level subsystem. An analysis system contains analysis subsystems, and/or analysis service packages, and/or analysis classes (i.e., entity, boundary, and control), and relationships.

An Analysis Subsystem is a subsystem containing other analysis subsystems, analysis service packages, analysis classes (i.e., entity, boundary, and control), and relationships.

An Analysis Service Package is a subsystem containing analysis classes (i.e., entity, boundary, and control) and relationships.

Design

A Design Model is a model whose top-level package is a design system.

A Design System is a top-level subsystem. A design system contains design subsystems, and/or design service packages, and/or design classes, and relationships.

A Design Subsystem is a subsystem containing other design subsystems, design service packages, design classes, and relationships.

A Design Service Package is a subsystem containing design classes and relationships.

Implementation

An Implementation Model is a model whose top-level package is an implementation system.

An Implementation System is a top-level package. An implementation system contains implementation subsystems, and/or components, and relationships.

An Implementation Subsystem is a package containing implementation subsystems, and/or components, and relationships.

Notation

Package stereotypes are indicated with stereotype keywords in guillemets («stereotype name»). There are no stereotyped icons for packages.

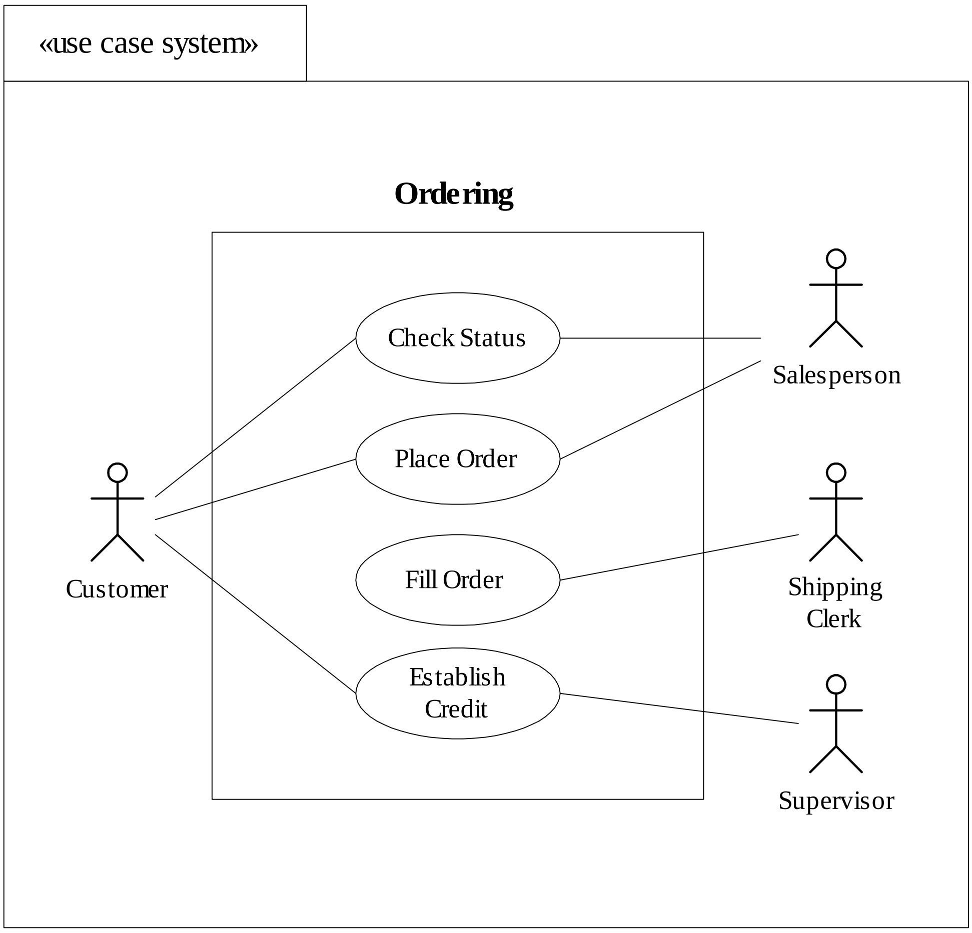


Figure 4-1 Objectory Packages

4.4.2 Class Stereotypes

Analysis classes come in the following three kinds: 1) entity, 2) control, and 3) boundary. Design classes are not stereotyped in the process.

Entity

Entity is a class that is passive; that is, it does not initiate interactions on its own. An entity object may participate in many different use case realizations and usually outlives any single interaction.

Control

Control is a class, an object of which denotes an entity that controls interactions between a collection of objects. A control class usually has behavior specific for one use case and a control object usually does not outlive the use case realizations in which it participates.

Boundary

A Boundary is a class that lies on the periphery of a system, but within it. It interacts with actors outside the system as well as objects of all three kinds of analysis classes within the system.

Notation

Class stereotypes can be shown with keywords in guillemets. They can also be shown with the following special icons.

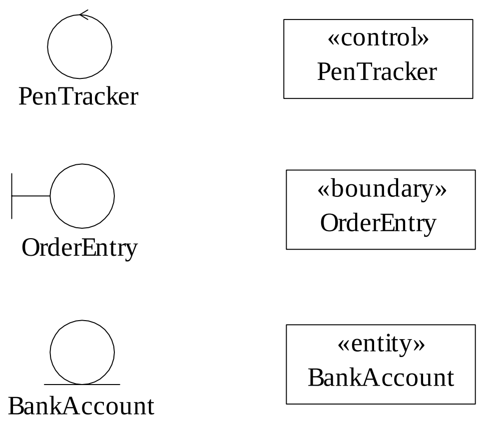


Figure 4-2 Class Stereotypes

4.4.3 Association Stereotypes

The following are special associations between classes.

Communicates

Communicates is an association between actors and use cases denoting that the actor sends messages to the use case and/or the use case sends messages to the actor. This may be one-way or two-way navigation. The direction of communication is indicated by the navigability of the association.

Subscribes

Subscribes is an association whose source is a class (called the subscriber) and whose target is a class (called the publisher). The subscriber specifies a set of events. The subscriber is notified when one of those events occurs in the target.

Notation

Association stereotypes are indicated by keywords in guillemets. There are no special stereotype icons. The stereotype «communicates» on Actor-Use Case associations may be omitted, since it is the only kind of relationships between actors and use cases.

4.5 Well-Formedness Rules

Stereotyped model elements are subject to certain constraints, in addition to the constraints imposed on all elements of their kind.

4.5.1 Generalization

All the modeling elements in a generalization must be of the same stereotype.

4.5.2 Association

Apart from standard UML combinations, the following combinations are allowed for each stereotype.

Table 4-2 Valid Association Stereotype Combinations

To: **From::**	**actor**	**boundary**	**entity**	**control**
actor		communicates		
boundary	communicates	communicates	communicates subscribes	communicates
entity			communicates subscribes	
control		communicates	communicates subscribes	communicates

Part 2 - UML Extension for Business Modeling

4.6 Introduction

The UML Extension for Business Modeling is defined in terms of the UML's extension mechanisms, namely Stereotypes, TaggedValues, and Constraints. See the UML Semantics chapter for a full description of the UML extension mechanisms.

This section describes stereotypes that can be used to tailor the use of UML for business modeling. All of the UML concepts can be used for business modeling, but providing business stereotypes for some common situations provides a common terminology for this domain. Note that UML can be used to model different kinds of systems (software systems, hardware systems, and real-world organizations). Business modeling models real-world organizations.

This section is not meant to be a complete definition of business modeling concepts and how to apply them, but it serves the purpose of registering this extension, including its icons.

4.7 Summary of Extension

4.7.1 Stereotypes

Table 4-3 Metamodel Class Stereotypes

Metamodel Class	Stereotype Name
Model	use case model
Package	use case system
Package	use case package
Model	object model
Subsystem	object system
Subsystem	organization unit
Subsystem	work unit
Class	worker
Class	case worker
Class	internal worker
Class	entity
Collaboration	use case realization
Association	subscribes

4.7.2 Tagged Values

This extension does not currently introduce any new TaggedValues.

4.7.3 Constraints

This extension does not currently introduce any new Constraints, other than those associated with the well-formedness semantics of the stereotypes introduced.

4.7.4 Prerequisite Extensions

This extension requires no other extensions to the UML for its definition.

4.8 Stereotypes and Notation

4.8.1 Model, Package, and Subsystem Stereotypes

A business system comprises several different, but related models. The models are characterized by being exterior or interior to the business system they represent. Exterior models are use case models and interior models are object models. A large business system may be partitioned into subordinate business systems. The following are the model stereotypes.

Use Case

A Use Case Model is a model that describes the business processes of a business and their interactions with external parties such as customers and partners.

A use case model describes:

- the businesses modeled as use cases.
- parties exterior to the business (e.g., customers and other businesses) modeled as actors.
- the relationships between the external parties and the business processes.

A Use Case System is the top-level package in a use case model. A use case system contains use case packages, use cases, actors, and relationships.

A Use Case Package is a package containing use cases and actors with relationships. A use case is not partitioned over several use case packages.

Object

An Object Model is a model in which the top-level package is an object system. These models describe the things interior to the business system itself.

An Object System is the top-level subsystem in an object model. An object system contains organization units, classes (workers, work units, and entities), and relationships.

Organization Unit

Organization Unit is a subsystem corresponding to an organization unit of the actual business. An organization unit subsystem contains organization units, work units, classes (workers and entities), and relationships.

Work Unit

A Work Unit is a subsystem that contains one or more entities.

A work unit is a task-oriented set of objects that form a recognizable whole to the end user. It may have a facade defining the view of the work unit's entities relevant to the task.

Notation

Package stereotypes are indicated with stereotype keywords in guillemets («stereotype name»). There are no special stereotyped icons for packages.

4.8.2 Class Stereotypes

Business objects come in the following kinds:

- actor (defined in the UML)
- worker
- case worker
- internal worker
- entity

Worker

A Worker is a class that represents an abstraction of a human that acts within the system. A worker interacts with other workers and manipulates entities while participating in use case realizations.

Case Worker

A Case Worker is a worker who interacts directly with actors outside the system.

Internal Worker

An Internal Worker is a worker that interacts with other workers and entities inside the system.

Entity

An Entity is a class that is passive; that is, it does not initiate interactions on its own. An entity object may participate in many different use case realizations and usually outlives any single interaction. In business modeling, entities represent objects that workers access, inspect, manipulate, produce, and so on. Entity objects provide the basis for sharing among workers participating in different use case realizations.

Notation

Class stereotypes can be shown with keywords in guillemets within the normal class symbol. They can also be shown with the following special icons.

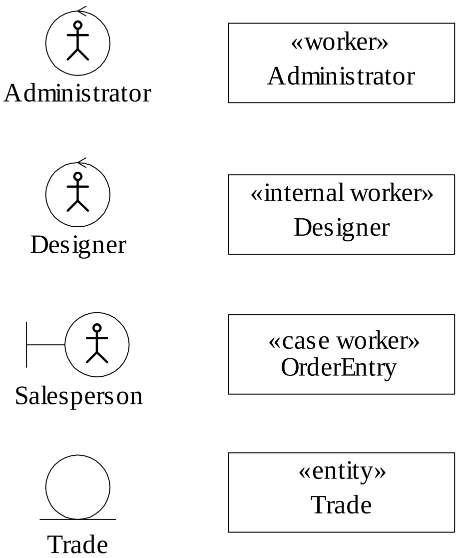


Figure 4-3 Class Stereotypes

The preceding icons represent common concepts useful in most business models.

Example of Alternate Notations

Tools and users are free to add additional icons to represent more specific concepts. Examples of such icons include icons for documents and actions, as shown in Figure 4-4.

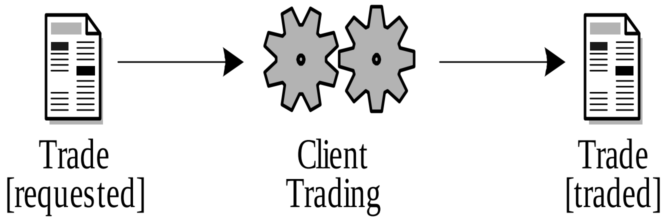


Figure 4-4 Example of Special Icons for Entities and Actions

In this example, "Trade [requested]" and "Trade [traded]" represent an entity in two states, where the Trade is the dominant entity of a Trade Document work unit. Client Trading is an action. The icons are designed to be meaningful in the particular problem domain.

4.8.3 Association Stereotypes

The following are special business modeling associations between classes:

Communicates

Communicates is an association used by two instances to interact. This may be one-way or two-way navigation. The direction of communication is the same as the navigability of the association.

Subscribes

Subscribes is an association whose source is a class (called the subscriber) and whose target is a class (called the publisher). The subscriber specifies a set of events. The subscriber is notified when one of those events occurs in the target.

Notation

Association stereotypes are indicated by keywords in guillemets. There are no special stereotype icons.

4.9 Well-Formedness Rules

Stereotyped model elements are subject to certain constraints in addition to the constraints imposed on all elements of their kind.

4.9.1 Generalization

All the modeling elements in a generalization must be of the same stereotype.

4.9.2 Association

Apart from standard UML combinations, the following combinations are allowed for each stereotype.

Table 4-4 Valid Association Stereotype Combinations

To: **From:**	**actor**	**case worker**	**entity**	**work unit**	**internal worker**
actor		communicates		communicatessubscribes	
case worker	communicates	communicates	communicatessubscribes	communicatessubscribes	communicates
entity			communicatessubscribes	communicates	
work unit	communicates	communicates	communicatessubscribes	communicatessubscribes	communicates
internal worker		communicates	communicatessubscribes	communicatessubscribes	communicates

OA&D CORBAfacility Interface Definition 5

This chapter specifies the interfaces for a CORBAfacility for Object Analysis & Design, consistent with the Unified Modeling Language, version 1.0. An OA&D Facility is a repository for models expressed in the UML. The facility enables the creation, storage, and manipulation of UML models. The facility enables clients to be developed that provide a wide variety of model-based development capabilities, including:

- Drawing and animation of UML models in UML and other notations
- Enforcement of process and method style guidelines
- Metrics, queries, and reports
- Automation of certain development lifecycle activities (e.g., through design wizards and code generation).

Contents

This chapter contains the following sections.

5

5.1 Service Description

There are two sets of interfaces provided: 1) generic and 2) tailored. Both sets of interfaces enable the creation and traversal of UML model elements. The generic interfaces are included in the Reflective module.

This is a set of general-purpose interfaces that provide utility for browser type functionality and as a base for the tailored interfaces. They are more fully described in the Meta-Object Facility (MOF) specification.

A set of tailored interfaces that are specifically typed to the UML metamodel elements is defined. The tailored interfaces inherit from the generic interfaces. The tailored interfaces provide capabilities necessary to instantiate, traverse, and modify UML model elements in the facility, directly in terms of the UML metamodel, with type safety. The specifications of the tailored interfaces were generated by applying a set of transformations to the UML semantic metamodel. Because the tailored interfaces were generated consistently from a set of patterns (described more fully in the MOF specification), they are easy to understand and program against. It is feasible to generate automatically the implementation for the OA&D facility, for the most part, because of these patterns and because the UML metamodel is strictly structural.

The UML is designed with a layered architecture. Implementers can choose which layers to implement, and whether to implement only the generic interfaces or the generic and tailored interfaces.

One of the primary goals was to advance the state of the industry by enabling OO modeling tool interoperability. This OA&D facility defines a set of interfaces to provide that tool interoperability. However, enabling meaningful exchange of model information between tools requires agreement on semantics and their visualization. The metamodel documenting the UML semantics and notation is defined in the UML Semantics chapter. Most of the IDL defined in this document is a direct mapping of the UML v1.0 metamodel, based on the IDL mapping defined in the MOF specification. Because the UML semantics are sufficiently complex, they are documented separately in the UML Semantics chapter, whereas this chapter is void of explanations of semantics.

5.1.1 Tool Sharing Options

A major goal is to achieve semantic interoperability between OA&D tools. Figure 5-1 depicts several viable alternatives to exchanging model information between tools.

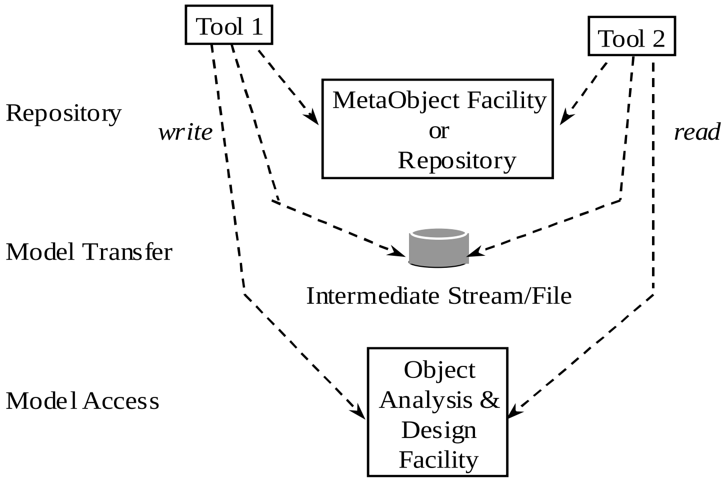


Figure 5-1 Model Sharing Alternatives between OA&D Tools

General-purpose Repository

Two tools could interface to the same repository and access a model there. The MetaObject Facility (MOF) could be this repository. This mapping is very implementation dependent, since the MOF cannot necessarily enforce the richer semantics defined in a UML-compliant tool. This approach is not described in this response, although the mapping to the MOF is described in the Preface.

Model Transfer

Two tools could understand the same stream format and exchange models via that stream, which could be a file. This is referred to as an "import facility." Having this interface would be necessary to provide a path for tools that are not implemented in an API (CORBA or non-CORBA) or repository environment. The Preface discusses stream format and CDIF further.

Model Access

Two tools could exchange models on a detail-by-detail basis. This is referred to as a "connection facility." Although this would not be the most efficient method for sharing an entire model, this type of access enables semantic interoperability to the greatest degree and is extremely useful for client applications. This, too, is a repository, but its interfaces are specific to the OA&D domain. A set of IDL interfaces is defined in this document to provide model access.

In summary, the OA&D Facility defines IDL interfaces for clients to use in a Model Access mode. The interface is consistent with the UML metamodel contained in this response.

5.2 Mapping of UML Semantics to Facility Interfaces

Understanding the process used to generate the IDL for this facility is helpful in understanding the resulting IDL. The process was as follows:

1. Converted the UML Semantics Metamodel into the Interface Metamodel, making necessary refinements for CORBA interfaces.
2. Stored the Interface Metamodel into a MetaObject Facility prototype as an instance of the MOF meta-metamodel elements.
3. Generated IDL from the MOF, based on the mapping defined in the MOF proposal.

5.2.1 Transformation of UML Semantics Metamodel into Interfaces Metamodel

A model was created representing the interfaces required on the OA&D Facility. This interface metamodel is nearly identical to the UML Semantics metamodel, so it is not documented explicitly. The following list summarizes the conversions made from the UML Semantics metamodel:

- Named associations and their ends, where names were missing.
- Deleted derived associations, since they would have resulted in redundant interfaces.
- Mapped all UML data types and select classes to CORBA data types.
- Transformed association classes into more fundamental structures. (A goal of the MOF was simplicity, so it does not support association classes.)
- Combined the UML DataTypes and Extension Mechanisms packages into Core, resulting in easier-to-use name scoping for the interfaces.
- Renamed certain classifiers, association ends, and attributes to avoid conflicts with words reserved in Reflective interfaces, CORBA, and MOF.
- Set navigability for associations to be uni-directional between classifiers that crossed packages. This was necessary to permit implementations of the more fundamental packages/modules without requiring a full UML implementation[1] [2] .

1. All other navigability is assumed to be useful. For example, although an implementation environment class should not know about its child classes, it is useful for an OA&D tool.

- Renamed AssociationClass to UmlAssociationClass. (The MOF IDL generation creates a FooClass for every Foo, so the UML class 'Association' would have created an 'AssociationClass' interface which would have clashed.)
- Renamed enumeration literal names so they would be unique within the resulting IDL modules.

The IDL generation from the MOF assures that all root classes in the interface metamodel are specializations of Reflective::RefObject, so this relationship is assumed to be present in the interface metamodel.

The remainder of this chapter describes the transformation of Association Classes as in the UML Semantics metamodel to the interface metamodel, summarizes the usage of CORBA data types, and summarizes the source and purpose of the MOF Reflective interfaces.

Transformation for Association Classes

Since the MOF does not represent the semantics of association classes directly, we needed to convert the association classes in the UML into a simple class and add necessary relationships to enable complete navigation (in the resulting facility IDL). Figure 5-2 shows an example association class as it would appear in the semantic metamodel.

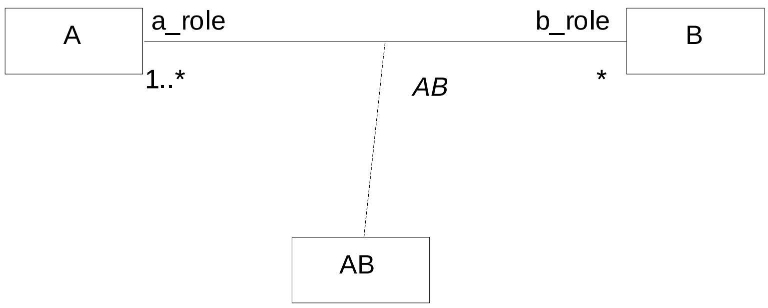


Figure 5-2 An Association Class in a Semantic Metamodel

2. The OA&D facility interfaces exclude navigation from classes in base packages/modules to classes in dependent packages/modules. This is deliberate. For example, an instance of a UML class does not know about a State Machine that might be attached to it. Vendors should consider adding interfaces to support such navigation when implementing multiple modules.

Figure 5-3 shows the corresponding transformed structure in the interface model.

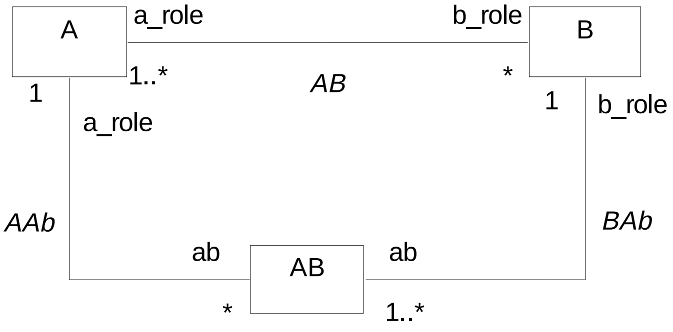


Figure 5-3 Corresponding Association Class in an Interface Metamodel

MOF Generic Interfaces

The MOF specification fully describes the generic interfaces. As a summary, the generic interfaces in the Reflective module provide the following:

- consistent treatment of type information,
- exception handling (including constraint violations, missing parameters, etc.), and
- generic creation and traversal of objects.

Note – The MOF specification replaces the definition of the Reflective module contained in this specification.

DataTypes for Interface

UML itself is platform independent; therefore, during the translation to the interface model, specific CORBA data types were selected as the structural base. These are listed in Table 5-1.

Table 5-1 Data Types

UML DataType	IDL Declaration
String	//string
Integer	typedef short Integer;
Uninterpreted	typedef any Uninterpreted;
Time	typedef float Time;
Name	struct Name { string body ; };
GraphicMarker	struct GraphicMarker { any body ; } ;
Geometry	struct Geometry { any body ; } ;
TimeExpression	struct TimeExpression { Name language ; any body ; } ;

Table 5-1 Data Types

ObjectSetExpression	struct ObjectSetExpression { Name language ; any body ; } ;
ProcedureExpression	struct ProcedureExpression { Name language ; any body ; } ;
Expression	struct Expression { Name language ; any body ; } ;
BooleanExpression	struct BooleanExpression { Name language ; any body ; } ;
Mapping	struct Mapping { any body ; } ;
MultiplicityRange	struct MultiplicityRange { lower short; upper short ; } ;
Multiplicity	sequence < MultiplicityRange > Multiplicity;
ChaneableKind	enum ChangeableKind { ck_none, ck_frozen, ck_addOnly };
OperationDirectionKind	enum OperationDirectionKind { odk_provide, odk_require };
ParameterDirectionKind	enum ParameterDirectionKind {pdk_in, pdk_inout, pdk_out, pdk_return};
MessageDirectionKind	enum MessageDirectionKind { mdk_activation, mdk_return };
SynchronousKind	enum SynchronousKind { sk_synchronous, sk_asynchronous};
ScopeKind	enum ScopeKind {sk_instance, sk_type};
VisibilityKind	enum VisibilityKind { vk_publc, vk_protected, vk_private };
PseudostateKind	enum PseudostateKind { pk_initial, pk_final, pk_shallowHistory, pk_deepHistory, pk_join, pk_fork, pk_branch, pk_or };
CallConcurrencyKind	enum CallConcurrencyKind { cck_sequential, cck_guarded, cck_concurrent } ;
AggregationKind	enum AggregationKind {ak_none, ak_shared, ak_composite};

5.2.2 *Mapping of Interface Model into MOF*

The UML metamodel elements can be expressed as instances of MOF meta-metamodel elements. This mapping is summarized in the table below for the relevant elements in the interface metamodel.

Table 5-2 Relevant Elements in the Interface Metamodel

Package	**Package, Contains**
Class	Class, Contains
Attribute	Attribute, Contains, IsOfType
DataType	DataType
Association	Association, AssociationEnd, Reference, Contains, RefersTo, IsOfType
Generalization	Generalizes

These mappings are relatively straightforward, with the exception of how an Association is instantiated. Figure 5-4 on page 5-8 shows an example association as it would appear in the interface model. Figure 5-5 on page 5-8 illustrates a relevant view of the MOF meta-metamodel.

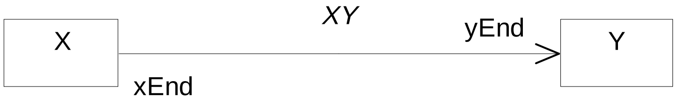


Figure 5-4 Association at Meta-Model Level Example

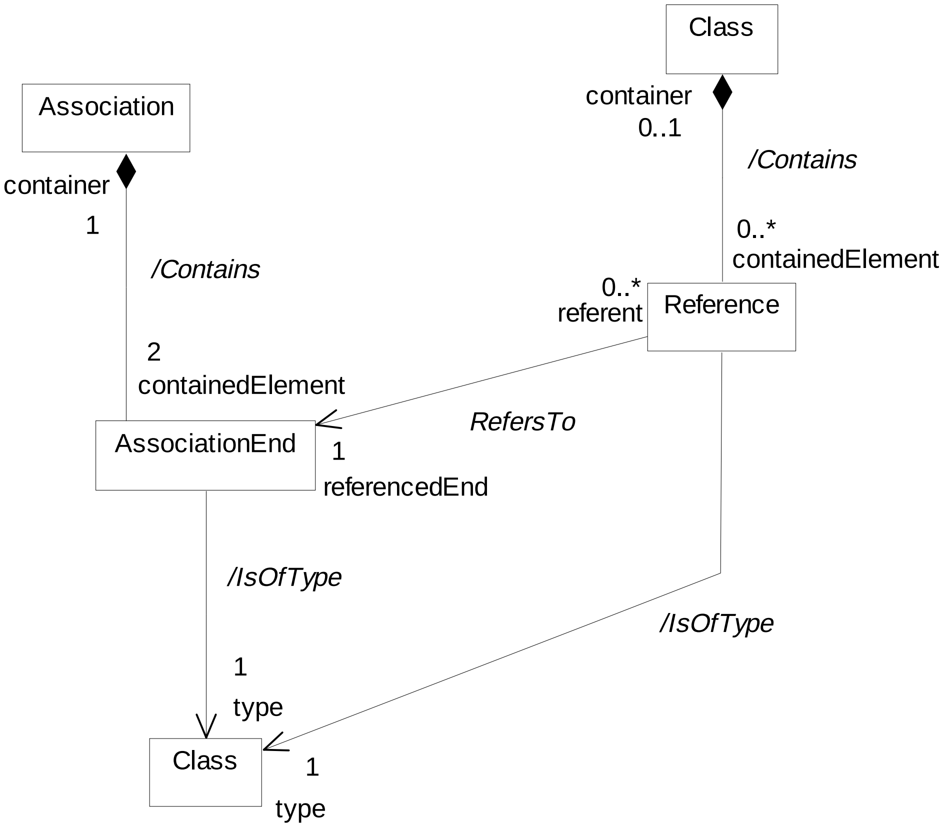


Figure 5-5 Projection of MOF Meta-Metamodel

Figure 5-6 on page 5-9 is a collaboration diagram showing how the association would be instantiated in terms of the MOF.

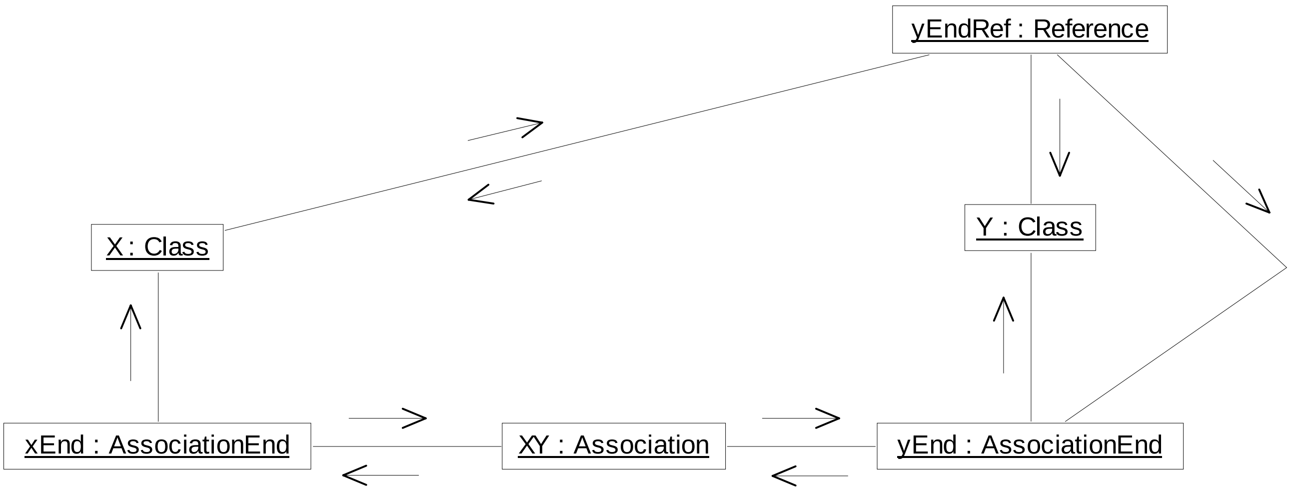


Figure 5-6 Collaboration Diagram showing Association Instantiated in Terms of MOF

In Figure 5-6, the message arrows are based on the navigation in the MOF meta-metamodel and indicate structural knowledge and potential messaging using the resulting interface.

5.2.3 Mapping from MOF to IDL

The description for the mapping from instances of models stored in the MOF is described in detail in the MOF specification. The result of this mapping is the generated IDL in this specification

5.3 Facility Implementation Requirements

Although this chapter focuses on defining the interfaces for the facility and leaves implementation decisions up to the creativity of vendors, there are some implementation requirements.

The UML Standard Elements (stereotypes, constraints, and tags) must be known to a facility implementation, or provided via a load. This is necessary so that the interoperability of these elements can be achieved. The semantics of the standard elements (e.g., containment restrictions) must be enforced. The Standard Elements are documented in the UML Semantics chapter.

The facility interfaces inherit from generic interfaces defined in the Reflective module. These interfaces provide common operations, such as verify(). The verify() operation should be implemented to return well-formedness violations numbered equal to the well-formedness rule following the meta-class definition in the UML Semantics chapter. This includes the semantics for the UML Standard Elements. The Reflective interfaces and exception handling is described in the Meta Object Facility (MOF) specification.

5.4 IDL Modules

5.4.1 Reflective

```
#ifndef REFLECTIVE_IDL
#define REFLECTIVE_IDL

// #include <orb.idl>
module Reflective {

  interface RefBaseObject;

  interface RefObject;
   typedef sequence < RefObject > RefObjectUList;
   typedef sequence < RefObject > RefObjectSet;

   interface RefAssociation;
   interface RefPackage;

   typedef RefObject DesignatorType;
   typedef any ValueType;
   typedef sequence < ValueType > ValueTypeList;
   typedef sequence < RefObject, 2 > Link;
   typedef sequence < ValueType > ErroneousValues;

   const string UNDERFLOW_VIOLATION = "underflow";
   const string OVERFLOW_VIOLATION = "overflow";
   const string DUPLICATE_VIOLATION = "duplicate";
   const string TYPE_CLOSURE_VIOLATION = "type closure";
   const string COMPOSITION_VIOLATION = "composition";
   const string INVALID_OBJECT_VIOLATION = "invalid object";

   struct StructuralViolation {
     string violation_kind;
     RefObject element_designator;
     ErroneousValues offending_values;
   };
```

```
typedef sequence < StructuralViolation > StructuralViolationSet;
exception StructuralError {
  StructuralViolationSet violations;
};
struct ConstraintViolation {
  RefObject constraint_designator;
  ErroneousValues offending_values;
  string explanation_text;
};
exception ConstraintError {
  ConstraintViolation violation;
};
struct ErrorDescription {
  string error_name;
  ErroneousValues offending_values;
  string explanation_text;
};
exception SemanticError {
  ErrorDescription error;
};

exception NotFound {};
exception NotSet {};
exception BadPosition {};
exception AlreadyCreated {};
exception InvalidLink {};
exception InvalidDesignator {
  DesignatorType designator;
  string element_kind;
};
exception InvalidValue {
  DesignatorType designator;
  string element_kind;
  ValueType value;
  CORBA::TypeCode type_expected;
};
exception InvalidObject {
  DesignatorType designator;
  RefObject obj;
  CORBA::TypeCode type_expected;
};
```

```
exception MissingParameter {
  DesignatorType designator;
};
exception TooManyParameters {};
exception OtherException {
  DesignatorType exception_designator;
  ValueTypeList exception_values;
};

interface RefBaseObject {
  DesignatorType meta_object ();
  boolean itself (in RefBaseObject other_object);
  RefBaseObject repository_container ();
}; // end of RefBaseObject


interface RefObject : RefBaseObject {
  boolean is_instance_of (in DesignatorType obj_type,
                 in boolean consider_subtypes);
  RefObject create_instance (in ValueTypeList args)
    raises (TooManyParameters,
            MissingParameter,
            InvalidValue,
            AlreadyCreated,
            StructuralError,
             ConstraintError,
        SemanticError);
  RefObjectSet all_objects (in boolean include_subtypes);
  void set_value (in DesignatorType feature,
                  in ValueType value)
    raises (InvalidDesignator,
            InvalidValue,
            StructuralError,
            ConstraintError,
             SemanticError);
  ValueType value (in DesignatorType feature)
    raises (InvalidDesignator,
            SemanticError);
  void add_value (in DesignatorType feature,
                  in ValueType value)
    raises (InvalidDesignator,
```

```
          InvalidValue,
          StructuralError,
          ConstraintError,
          SemanticError);
void add_value_before (in DesignatorType feature,
                       in ValueType value,
                       in ValueType existing_value)
  raises (InvalidDesignator,
          InvalidValue,
          NotFound,
          StructuralError,
          ConstraintError,
          SemanticError);
void add_value_at (in DesignatorType feature,
                   in ValueType value,
                   in long position)
  raises (InvalidDesignator,
          InvalidValue,
          BadPosition,
          StructuralError,
          ConstraintError,
          SemanticError);
void modify_value (in DesignatorType feature,
                   in ValueType existing_value,
                   in ValueType new_value)
  raises (InvalidDesignator,
          InvalidValue,
          NotFound,
          StructuralError,
          ConstraintError,
          SemanticError);
void modify_value_at (in DesignatorType feature,
                      in ValueType new_value,
                      in long position)
  raises (InvalidDesignator,
          InvalidValue,
          BadPosition,
          StructuralError,
          ConstraintError,
          SemanticError);
void remove_value (in DesignatorType feature,
```

```
                     in ValueType existing_value)
    raises (InvalidDesignator,
            InvalidValue,
            NotFound,
            StructuralError,
            ConstraintError,
            SemanticError);
  void remove_value_at (in DesignatorType feature,
                        in long position)
    raises (InvalidDesignator,
            InvalidValue,
            BadPosition,
            NotFound,
            StructuralError,
            ConstraintError,
            SemanticError);
  ValueType invoke_operation (in DesignatorType requested_operation,
                              in ValueTypeList args)
    raises (InvalidDesignator,
            TooManyParameters,
            MissingParameter,
            InvalidValue,
            OtherException,
            ConstraintError,
            SemanticError);
}; // end of interface RefObject

interface RefAssociation : RefBaseObject {
  boolean link_exists (in Link some_link)
    raises (InvalidLink,
            SemanticError);
  RefObjectUList query (in DesignatorType query_end,
                        in RefObject query_object)
    raises (InvalidDesignator,
            InvalidObject,
            SemanticError);
  void add_link (in Link new_link)
    raises (InvalidLink,
            StructuralError,
            ConstraintError,
            SemanticError);
```

```
    void add_link_before (in Link new_link,
                          in DesignatorType position_end,
                          in RefObject position_value)
      raises (InvalidDesignator,
              InvalidObject,
              InvalidLink,
              NotFound,
              StructuralError,
              ConstraintError,
              SemanticError);
    void modify_link (in Link existing_link,
                      in DesignatorType position_end,
                      in RefObject position_value)
      raises (InvalidDesignator,
              InvalidObject,
              InvalidLink,
              NotFound,
              StructuralError,
              ConstraintError,
              SemanticError);
    void remove_link (in Link existing_link)
      raises (InvalidLink,
              NotFound,
              StructuralError,
              ConstraintError,
              SemanticError);
  }; // end of interface RefAssociation

  interface RefPackage : RefBaseObject {
    RefObject get_class_ref (in DesignatorType type)
      raises (InvalidDesignator);
    RefAssociation get_association (in DesignatorType association)
      raises (InvalidDesignator);
    RefPackage get_nested_package (in DesignatorType nested_package)
      raises (InvalidDesignator);
  }; // end of interface RefPackage
}; // end of module Reflective

#endif
```

UMLCore

```
#include "Reflective.idl"

module UmlCore {
   interface UmlCorePackage;
   interface Enumeration;
   interface EnumerationClass;
   typedef sequence<Enumeration> EnumerationUList;
   interface Generalization;
   interface GeneralizationClass;
   typedef sequence<Generalization> GeneralizationUList;
   typedef sequence<Generalization> GeneralizationSet;
   interface Class;
   interface ClassClass;
   typedef sequence<Class> ClassUList;
   interface Dependency;
   interface DependencyClass;
   typedef sequence<Dependency> DependencyUList;
   typedef sequence<Dependency> DependencySet;
   interface Parameter;
   interface ParameterClass;
   typedef sequence<Parameter> ParameterUList;
   interface GeneralizableElement;
   interface GeneralizableElementClass;
   typedef sequence<GeneralizableElement> GeneralizableElementUList;
   interface Constraint;
   interface ConstraintClass;
   typedef sequence<Constraint> ConstraintUList;
   typedef sequence<Constraint> ConstraintSet;
   interface ModelElement;
   interface ModelElementClass;
   typedef sequence<ModelElement> ModelElementUList;
   typedef sequence<ModelElement> ModelElementSet;
   interface ElementOwnership;
   interface ElementOwnershipClass;
   typedef sequence<ElementOwnership> ElementOwnershipUList;
   typedef sequence<ElementOwnership> ElementOwnershipSet;
   interface Classifier;
   interface ClassifierClass;
   typedef sequence<Classifier> ClassifierUList;
   typedef sequence<Classifier> ClassifierSet;
   interface UmlAttribute;
```

```
interface UmlAttributeClass;
typedef sequence<UmlAttribute> UmlAttributeUList;
interface EnumerationLiteral;
interface EnumerationLiteralClass;
typedef sequence<EnumerationLiteral> EnumerationLiteralUList;
interface Element;
interface ElementClass;
typedef sequence<Element> ElementUList;
interface Namespace;
interface NamespaceClass;
typedef sequence<Namespace> NamespaceUList;
interface Primitive;
interface PrimitiveClass;
typedef sequence<Primitive> PrimitiveUList;
interface UmlAssociationClass;
interface UmlAssociationClassClass;
typedef sequence<UmlAssociationClass> UmlAssociationClassUList;
interface StructuralFeature;
interface StructuralFeatureClass;
typedef sequence<StructuralFeature> StructuralFeatureUList;
interface Feature;
interface FeatureClass;
typedef sequence<Feature> FeatureUList;
typedef sequence<Feature> FeatureSet;
interface Stereotype;
interface StereotypeClass;
typedef sequence<Stereotype> StereotypeUList;
typedef sequence<Stereotype> StereotypeSet;
interface Association;
interface AssociationClass;
typedef sequence<Association> AssociationUList;
interface TaggedValue;
interface TaggedValueClass;
typedef sequence<TaggedValue> TaggedValueUList;
typedef sequence<TaggedValue> TaggedValueSet;
interface AssociationEnd;
interface AssociationEndClass;
typedef sequence<AssociationEnd> AssociationEndUList;
typedef sequence<AssociationEnd> AssociationEndSet;
interface Operation;
interface OperationClass;
```

```
typedef sequence<Operation> OperationUList;
interface BehavioralFeature;
interface BehavioralFeatureClass;
typedef sequence<BehavioralFeature> BehavioralFeatureUList;
typedef sequence<BehavioralFeature> BehavioralFeatureSet;
interface Request;
interface RequestClass;
typedef sequence<Request> RequestUList;
interface Method;
interface MethodClass;
typedef sequence<Method> MethodUList;
typedef sequence<Method> MethodSet;
interface DataType;
interface DataTypeClass;
typedef sequence<DataType> DataTypeUList;
interface UmlInterface;
interface UmlInterfaceClass;
typedef sequence<UmlInterface> UmlInterfaceUList;
interface Structure;
interface StructureClass;
typedef sequence<Structure> StructureUList;
enum AggregationKind {ak_none, ak_shared, ak_composite};
enum CallConcurrencyKind { cck_sequential, cck_guarded, cck_concurrent };
enum ChangeableKind { ck_none, ck_frozen, ck_addOnly };
typedef short Integer;
enum MessageDirectionKind { mdk_activation, mdk_return };
enum OperationDirectionKind { odk_provide, odk_require };
enum ParameterDirectionKind {pdk_in, pdk_inout, pdk_out, pdk_return};
enum ScopeKind {sk_instance, sk_type};
enum SynchronousKind { sk_synchronous, sk_asynchronous };
typedef float Time;
typedef any Uninterpreted;
enum VisibilityKind { vk_publc, vk_protected, vk_private };
struct Name { string body; };
struct Expression { Name language; any body; };
struct BooleanExpression { Name language; any body; };
struct ObjectSetExpression { Name language; any body; };
struct ProcedureExpression { Name language; any body; };
struct TimeExpression { Name language; any body; };
struct Geometry { any body; };
struct GraphicMarker { any body; };
```

```
struct MultiplicityRange { short lower; short upper; };
enum PseudostateKind { pk_initial,
                       pk_final,
                       pk_shallowHistory,
                       pk_deepHistory,
                       pk_join,
                       pk_fork,
                       pk_branch,
                       pk_or };
typedef sequence <MultiplicityRange> Multiplicity;
struct Mapping { any body; };

interface ElementClass : Reflective::RefObject {
   readonly attribute ElementUList all_of_kind_element;
};

interface Element : ElementClass  { };

interface TaggedValueClass : ElementClass {
   readonly attribute TaggedValueUList all_of_kind_tagged_value;
   readonly attribute TaggedValueUList all_of_type_tagged_value;
   TaggedValue create_tagged_value (in Name tag,
                                    in Uninterpreted uml_value)
      raises (Reflective::SemanticError);
};

interface TaggedValue : TaggedValueClass, Element {
   Name tag ()
      raises (Reflective::SemanticError);
   void set_tag (in Name new_value)
      raises (Reflective::SemanticError);
   Uninterpreted uml_value ()
      raises (Reflective::SemanticError);
   void set_uml_value (in Uninterpreted new_value)
      raises (Reflective::SemanticError);
};

interface EnumerationLiteralClass : ElementClass {
   readonly attribute EnumerationLiteralUList
      all_of_kind_enumeration_literal;
```

```
      readonly attribute EnumerationLiteralUList
          all_of_type_enumeration_literal;
      EnumerationLiteral create_enumeration_literal (in UmlCore::Name name)
         raises (Reflective::SemanticError);
   };

   interface EnumerationLiteral : EnumerationLiteralClass, Element {
      UmlCore::Name name ()
         raises (Reflective::SemanticError);
      void set_name (in UmlCore::Name new_value)
         raises (Reflective::SemanticError);
      UmlCore::Enumeration enumeration ()
         raises (Reflective::SemanticError);
      void set_enumeration (in UmlCore::Enumeration new_value)
         raises (Reflective::SemanticError);
   };

   interface ModelElementClass : ElementClass {
      readonly attribute ModelElementUList all_of_kind_model_element;
   };

   interface ModelElement : ModelElementClass, Element {
      UmlCore::Name name ()
         raises (Reflective::SemanticError);
      void set_name (in UmlCore::Name new_value)
         raises (Reflective::SemanticError);
      UmlCore::Namespace namespace ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void set_namespace (in UmlCore::Namespace new_value)
         raises (Reflective::SemanticError);
      void unset_namespace ()
         raises (Reflective::SemanticError);
      DependencySet provision ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_provision (in DependencySet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_provision ()
         raises (Reflective::SemanticError);
      UmlCore::TaggedValueSet tagged_value ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_tagged_value (in UmlCore::TaggedValueSet new_value)
```

```
            raises (Reflective::StructuralError, Reflective::SemanticError);
        void remove_tagged_value ()
            raises (Reflective::SemanticError);
        UmlCore::ConstraintSet constraint ()
            raises (Reflective::NotSet, Reflective::SemanticError);
        void add_constraint (in UmlCore::ConstraintSet new_value)
            raises (Reflective::StructuralError, Reflective::SemanticError);
        void remove_constraint ()
            raises (Reflective::SemanticError);
        DependencySet requirement ()
            raises (Reflective::NotSet, Reflective::SemanticError);
        void add_requirement (in DependencySet new_value)
            raises (Reflective::StructuralError, Reflective::SemanticError);
        void remove_requirement ()
            raises (Reflective::SemanticError);
        ModelElement template ()
            raises (Reflective::NotSet, Reflective::SemanticError);
        void set_template (in ModelElement new_value)
            raises (Reflective::SemanticError);
        void unset_template ()
            raises (Reflective::SemanticError);
        ModelElementUList template_parameter ()
            raises (Reflective::NotSet, Reflective::SemanticError);
        void add_template_parameter (in ModelElementUList new_value)
            raises (Reflective::StructuralError, Reflective::SemanticError);
        void add_template_parameter_before (in ModelElement new_value,
                                            in ModelElement before)
            raises (Reflective::StructuralError,
                    Reflective::NotFound,
                    Reflective::SemanticError);
        void remove_template_parameter ()
            raises (Reflective::SemanticError);
        ElementOwnership namespace1 ()
            raises (Reflective::NotSet, Reflective::SemanticError);
        void set_namespace1 (in ElementOwnership new_value)
            raises (Reflective::SemanticError);
        void unset_namespace1 ()
            raises (Reflective::SemanticError);
    };

    interface FeatureClass : ModelElementClass {
```

```
   readonly attribute FeatureUList all_of_kind_feature;
};

interface Feature : FeatureClass, ModelElement {
   ScopeKind owner_scope ()
      raises (Reflective::SemanticError);
   void set_owner_scope (in ScopeKind new_value)
      raises (Reflective::SemanticError);
   VisibilityKind visibility ()
      raises (Reflective::SemanticError);
   void set_visibility (in VisibilityKind new_value)
      raises (Reflective::SemanticError);
   Classifier owner ()
      raises (Reflective::SemanticError);
   void set_owner (in Classifier new_value)
      raises (Reflective::SemanticError);
};

interface GeneralizationClass : ModelElementClass {
   readonly attribute GeneralizationUList all_of_kind_generalization;
   readonly attribute GeneralizationUList all_of_type_generalization;
   Generalization create_generalization (in UmlCore::Name name,
                                          in UmlCore::Name discriminator)
      raises (Reflective::SemanticError);
};

interface Generalization : GeneralizationClass, ModelElement {
   UmlCore::Name discriminator ()
      raises (Reflective::SemanticError);
   void set_discriminator (in UmlCore::Name new_value)
      raises (Reflective::SemanticError);
   GeneralizableElement subtype ()
      raises (Reflective::SemanticError);
   void set_subtype (in GeneralizableElement new_value)
      raises (Reflective::SemanticError);
   GeneralizableElement supertype ()
      raises (Reflective::SemanticError);
   void set_supertype (in GeneralizableElement new_value)
      raises (Reflective::SemanticError);
};
```

```
interface NamespaceClass : ModelElementClass {
   readonly attribute NamespaceUList all_of_kind_namespace;
   readonly attribute NamespaceUList all_of_type_namespace;
   Namespace create_namespace (in UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Namespace : NamespaceClass, ModelElement {
   ModelElementSet owned_element ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_owned_element (in ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_owned_element ()
      raises (Reflective::SemanticError);
   UmlCore::ElementOwnershipSet element_ownership ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_element_ownership (in UmlCore::ElementOwnershipSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_element_ownership ()
      raises (Reflective::SemanticError);
};

interface ParameterClass : ModelElementClass {
   readonly attribute ParameterUList all_of_kind_parameter;
   readonly attribute ParameterUList all_of_type_parameter;
   Parameter create_parameter (in UmlCore::Name name,
                               in Expression default_value,
                               in ParameterDirectionKind kind)
      raises (Reflective::SemanticError);
};

interface Parameter : ParameterClass, ModelElement {
   Expression default_value ()
      raises (Reflective::SemanticError);
   void set_default_value (in Expression new_value)
      raises (Reflective::SemanticError);
   ParameterDirectionKind kind ()
      raises (Reflective::SemanticError);
   void set_kind (in ParameterDirectionKind new_value)
      raises (Reflective::SemanticError);
   UmlCore::BehavioralFeature behavioral_feature ()
```

```
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_behavioral_feature (in UmlCore::BehavioralFeature new_value)
      raises (Reflective::SemanticError);
   void unset_behavioral_feature ()
      raises (Reflective::SemanticError);
   Classifier type ()
      raises (Reflective::SemanticError);
   void set_type (in Classifier new_value)
      raises (Reflective::SemanticError);
};

interface ConstraintClass : ModelElementClass {
   readonly attribute ConstraintUList all_of_kind_constraint;
   readonly attribute ConstraintUList all_of_type_constraint;
   Constraint create_constraint (in UmlCore::Name name,
                                 in BooleanExpression body)
      raises (Reflective::SemanticError);
};

interface Constraint : ConstraintClass, ModelElement {
   BooleanExpression body ()
      raises (Reflective::SemanticError);
   void set_body (in BooleanExpression new_value)
      raises (Reflective::SemanticError);
   ModelElementUList constrained_element ()
      raises (Reflective::SemanticError);
   void add_constrained_element (in ModelElementUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_constrained_element_before (in ModelElement new_value,
                                        in ModelElement before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_constrained_element ()
      raises (Reflective::SemanticError);
};

interface DependencyClass : ModelElementClass {
   readonly attribute DependencyUList all_of_kind_dependency;
};
```

```
interface Dependency : DependencyClass, ModelElement {
   string description ()
      raises (Reflective::SemanticError);
   void set_description (in string new_value)
      raises (Reflective::SemanticError);
   ModelElementSet supplier ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_supplier (in ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_supplier ()
      raises (Reflective::SemanticError);
   ModelElementSet client ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_client (in ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_client ()
      raises (Reflective::SemanticError);
};

interface RequestClass : ModelElementClass {
   readonly attribute RequestUList all_of_kind_request;
   readonly attribute RequestUList all_of_type_request;
   Request create_request (in UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Request : RequestClass, ModelElement { };

interface GeneralizableElementClass : UmlCore::NamespaceClass {
   readonly attribute GeneralizableElementUList
     all_of_kind_generalizable_element;
};

interface GeneralizableElement : GeneralizableElementClass,
                                 UmlCore::Namespace {
   boolean is_root ()
      raises (Reflective::SemanticError);
   void set_is_root (in boolean new_value)
      raises (Reflective::SemanticError);
   boolean is_leaf ()
      raises (Reflective::SemanticError);
```

```
    void set_is_leaf (in boolean new_value)
       raises (Reflective::SemanticError);
    boolean is_abstract ()
       raises (Reflective::SemanticError);
    void set_is_abstract (in boolean new_value)
       raises (Reflective::SemanticError);
    UmlCore::GeneralizationSet generalization ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_generalization (in UmlCore::GeneralizationSet new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_generalization ()
       raises (Reflective::SemanticError);
    UmlCore::GeneralizationSet specialization ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_specialization (in UmlCore::GeneralizationSet new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_specialization ()
       raises (Reflective::SemanticError);
};

interface BehavioralFeatureClass : FeatureClass {
    readonly attribute BehavioralFeatureUList
        all_of_kind_behavioral_feature;
};

interface BehavioralFeature : BehavioralFeatureClass, Feature {
    boolean is_query ()
       raises (Reflective::SemanticError);
    void set_is_query (in boolean new_value)
       raises (Reflective::SemanticError);
    UmlCore::ParameterUList parameter ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_parameter (in UmlCore::ParameterUList new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_parameter_before (in UmlCore::Parameter new_value,
                               in UmlCore::Parameter before)
       raises (Reflective::StructuralError,
               Reflective::NotFound,
               Reflective::SemanticError);
    void remove_parameter ()
       raises (Reflective::SemanticError);
```

```
};

interface ClassifierClass : GeneralizableElementClass {
   readonly attribute ClassifierUList all_of_kind_classifier;
};

interface Classifier : ClassifierClass, GeneralizableElement {
   UmlCore::FeatureUList feature ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_feature (in UmlCore::FeatureUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_feature_before (in UmlCore::Feature new_value,
                            in UmlCore::Feature before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_feature ()
      raises (Reflective::SemanticError);
   StructuralFeatureUList features ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_features (in StructuralFeatureUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_features_before (in StructuralFeature new_value,
                             in StructuralFeature before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_features ()
      raises (Reflective::SemanticError);
   UmlCore::ParameterUList parameter ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_parameter (in UmlCore::ParameterUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_parameter_before (in UmlCore::Parameter new_value,
                              in UmlCore::Parameter before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_parameter ()
      raises (Reflective::SemanticError);
   UmlCore::AssociationEndSet participant ()
```

```
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_participant (in UmlCore::AssociationEndSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_participant ()
      raises (Reflective::SemanticError);
   ClassifierSet realization ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_realization (in ClassifierSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_realization ()
      raises (Reflective::SemanticError);
   ClassifierSet specification ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_specification (in ClassifierSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_specification ()
      raises (Reflective::SemanticError);
   UmlCore::AssociationEndSet association_end ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_association_end (in UmlCore::AssociationEndSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_association_end ()
      raises (Reflective::SemanticError);
};

interface OperationClass : BehavioralFeatureClass {
   readonly attribute OperationUList all_of_kind_operation;
   readonly attribute OperationUList all_of_type_operation;
   Operation create_operation (in UmlCore::Name name,
                               in ScopeKind owner_scope,
                               in VisibilityKind visibility,
                               in boolean is_query,
                               in Uninterpreted specification,
                               in boolean is_polymorphic,
                               in CallConcurrencyKind concurrency)
      raises (Reflective::SemanticError);
};

interface Operation : OperationClass, BehavioralFeature {
   Uninterpreted specification ()
      raises (Reflective::SemanticError);
```

```
   void set_specification (in Uninterpreted new_value)
      raises (Reflective::SemanticError);
   boolean is_polymorphic ()
      raises (Reflective::SemanticError);
   void set_is_polymorphic (in boolean new_value)
      raises (Reflective::SemanticError);
   CallConcurrencyKind concurrency ()
      raises (Reflective::SemanticError);
   void set_concurrency (in CallConcurrencyKind new_value)
      raises (Reflective::SemanticError);
   UmlCore::MethodSet method ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_method (in UmlCore::MethodSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_method ()
      raises (Reflective::SemanticError);
};

interface StereotypeClass : GeneralizableElementClass {
   readonly attribute StereotypeUList all_of_kind_stereotype;
   readonly attribute StereotypeUList all_of_type_stereotype;
   Stereotype create_stereotype (in UmlCore::Name name,
                                 in boolean is_root,
                                 in boolean is_leaf,
                                 in boolean is_abstract,
                                 in Geometry icon)
      raises (Reflective::SemanticError);
};

interface Stereotype : StereotypeClass, GeneralizableElement {
   Geometry icon ()
      raises (Reflective::SemanticError);
   void set_icon (in Geometry new_value)
      raises (Reflective::SemanticError);
   UmlCore::TaggedValueSet required_tag ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_required_tag (in UmlCore::TaggedValueSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_required_tag ()
      raises (Reflective::SemanticError);
   ModelElementSet extended_element ()
```

```
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_extended_element (in ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_extended_element ()
      raises (Reflective::SemanticError);
   UmlCore::ConstraintSet stereotype_constraint ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_stereotype_constraint (in UmlCore::ConstraintSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_stereotype_constraint ()
      raises (Reflective::SemanticError);
};

interface StructuralFeatureClass : FeatureClass {
   readonly attribute StructuralFeatureUList
      all_of_kind_structural_feature;
};

interface StructuralFeature : StructuralFeatureClass, Feature {
   UmlCore::Multiplicity multiplicity ()
      raises (Reflective::SemanticError);
   void set_multiplicity (in UmlCore::Multiplicity new_value)
      raises (Reflective::SemanticError);
   ChangeableKind changeable ()
      raises (Reflective::SemanticError);
   void set_changeable (in ChangeableKind new_value)
      raises (Reflective::SemanticError);
   ScopeKind target_scope ()
      raises (Reflective::SemanticError);
   void set_target_scope (in ScopeKind new_value)
      raises (Reflective::SemanticError);
   Classifier type ()
      raises (Reflective::SemanticError);
   void set_type (in Classifier new_value)
      raises (Reflective::SemanticError);
};

interface DataTypeClass : ClassifierClass {
   readonly attribute DataTypeUList all_of_kind_data_type;
   readonly attribute DataTypeUList all_of_type_data_type;
   DataType create_data_type (in UmlCore::Name name,
```

```
                              in boolean is_root,
                              in boolean is_leaf,
                              in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface DataType : DataTypeClass, Classifier { };

interface UmlInterfaceClass : ClassifierClass {
   readonly attribute UmlInterfaceUList all_of_kind_uml_interface;
   readonly attribute UmlInterfaceUList all_of_type_uml_interface;
   UmlInterface create_uml_interface (in UmlCore::Name name,
                                      in boolean is_root,
                                      in boolean is_leaf,
                                      in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface UmlInterface : UmlInterfaceClass, Classifier { };

interface UmlAttributeClass : StructuralFeatureClass {
   readonly attribute UmlAttributeUList all_of_kind_uml_attribute;
   readonly attribute UmlAttributeUList all_of_type_uml_attribute;
   UmlAttribute create_uml_attribute (in UmlCore::Name name,
                                      in ScopeKind owner_scope,
                                      in VisibilityKind visibility,
                                  in UmlCore::Multiplicity multiplicity,
                                      in ChangeableKind changeable,
                                      in ScopeKind target_scope,
                                      in Expression initial_value)
      raises (Reflective::SemanticError);
};

interface UmlAttribute : UmlAttributeClass, StructuralFeature {
   Expression initial_value ()
      raises (Reflective::SemanticError);
   void set_initial_value (in Expression new_value)
      raises (Reflective::SemanticError);
   UmlCore::AssociationEnd association_end ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_association_end (in UmlCore::AssociationEnd new_value)
```

```
      raises (Reflective::SemanticError);
   void unset_association_end ()
      raises (Reflective::SemanticError);
};

interface AssociationEndClass : ModelElementClass {
   readonly attribute AssociationEndUList all_of_kind_association_end;
   readonly attribute AssociationEndUList all_of_type_association_end;
   AssociationEnd create_association_end (
                                       in UmlCore::Name name,
                                       in boolean is_navigable,
                                       in boolean is_ordered,
                                       in AggregationKind aggregation,
                                       in ScopeKind target_scope,
                                  in UmlCore::Multiplicity multiplicity,
                                       in ChangeableKind changeable)
      raises (Reflective::SemanticError);
};

interface AssociationEnd : AssociationEndClass, ModelElement {
   boolean is_navigable ()
      raises (Reflective::SemanticError);
   void set_is_navigable (in boolean new_value)
      raises (Reflective::SemanticError);
   boolean is_ordered ()
      raises (Reflective::SemanticError);
   void set_is_ordered (in boolean new_value)
      raises (Reflective::SemanticError);
   AggregationKind aggregation ()
      raises (Reflective::SemanticError);
   void set_aggregation (in AggregationKind new_value)
      raises (Reflective::SemanticError);
   ScopeKind target_scope ()
      raises (Reflective::SemanticError);
   void set_target_scope (in ScopeKind new_value)
      raises (Reflective::SemanticError);
   UmlCore::Multiplicity multiplicity ()
      raises (Reflective::SemanticError);
   void set_multiplicity (in UmlCore::Multiplicity new_value)
      raises (Reflective::SemanticError);
   ChangeableKind changeable ()
```

```
         raises (Reflective::SemanticError);
      void set_changeable (in ChangeableKind new_value)
         raises (Reflective::SemanticError);
      UmlCore::Association association ()
         raises (Reflective::SemanticError);
      void set_association (in UmlCore::Association new_value)
         raises (Reflective::SemanticError);
      UmlAttributeUList qualifier ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_qualifier (in UmlAttributeUList new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void add_qualifier_before (in UmlAttribute new_value,
                                 in UmlAttribute before)
         raises (Reflective::StructuralError,
                 Reflective::NotFound,
                 Reflective::SemanticError);
      void remove_qualifier ()
         raises (Reflective::SemanticError);
      Classifier type2 ()
         raises (Reflective::SemanticError);
      void set_type2 (in Classifier new_value)
         raises (Reflective::SemanticError);
      ClassifierSet specification ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_specification (in ClassifierSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_specification ()
         raises (Reflective::SemanticError);
   };

   interface AssociationClass : GeneralizableElementClass {
      readonly attribute AssociationUList all_of_kind_association;
      readonly attribute AssociationUList all_of_type_association;
      Association create_association (in UmlCore::Name name,
                                      in boolean is_root,
                                      in boolean is_leaf,
                                      in boolean is_abstract)
         raises (Reflective::SemanticError);
   };

   interface Association : AssociationClass, GeneralizableElement {
```

```
   AssociationEndUList connection ()
      raises (Reflective::SemanticError);
   void add_connection (in AssociationEndUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_connection_before (in AssociationEnd new_value,
                               in AssociationEnd before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void modify_connection (in AssociationEnd old_value,
                           in AssociationEnd new_value)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_connection ()
      raises (Reflective::StructuralError, Reflective::SemanticError);
};

interface MethodClass : BehavioralFeatureClass {
   readonly attribute MethodUList all_of_kind_method;
   readonly attribute MethodUList all_of_type_method;
   Method create_method (in UmlCore::Name name,
                         in ScopeKind owner_scope,
                         in VisibilityKind visibility,
                         in boolean is_query,
                         in ProcedureExpression body)
      raises (Reflective::SemanticError);
};

interface Method : MethodClass, BehavioralFeature {
   ProcedureExpression body ()
      raises (Reflective::SemanticError);
   void set_body (in ProcedureExpression new_value)
      raises (Reflective::SemanticError);
   Operation specification ()
      raises (Reflective::SemanticError);
   void set_specification (in Operation new_value)
      raises (Reflective::SemanticError);
};

interface EnumerationClass : DataTypeClass {
```

```
   readonly attribute EnumerationUList all_of_kind_enumeration;
   readonly attribute EnumerationUList all_of_type_enumeration;
   Enumeration create_enumeration (in UmlCore::Name name,
                                   in boolean is_root,
                                   in boolean is_leaf,
                                   in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Enumeration : EnumerationClass, DataType {
   EnumerationLiteralUList literal ()
      raises (Reflective::SemanticError);
   void add_literal (in EnumerationLiteralUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_literal_before (in EnumerationLiteral new_value,
                            in EnumerationLiteral before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_literal ()
      raises (Reflective::SemanticError);
};

interface ClassClass : ClassifierClass {
   readonly attribute ClassUList all_of_kind_class;
   readonly attribute ClassUList all_of_type_class;
   Class create_class (in UmlCore::Name name,
                       in boolean is_root,
                       in boolean is_leaf,
                       in boolean is_abstract,
                       in boolean is_active)
      raises (Reflective::SemanticError);
};

interface Class : ClassClass, Classifier {
   boolean is_active ()
      raises (Reflective::SemanticError);
   void set_is_active (in boolean new_value)
      raises (Reflective::SemanticError);
};
```

```
interface PrimitiveClass : DataTypeClass {
   readonly attribute PrimitiveUList all_of_kind_primitive;
   readonly attribute PrimitiveUList all_of_type_primitive;
   Primitive create_primitive (in UmlCore::Name name,
                               in boolean is_root,
                               in boolean is_leaf,
                               in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Primitive : PrimitiveClass, DataType { };

interface StructureClass : DataTypeClass {
   readonly attribute StructureUList all_of_kind_structure;
   readonly attribute StructureUList all_of_type_structure;
   Structure create_structure (in UmlCore::Name name,
                               in boolean is_root,
                               in boolean is_leaf,
                               in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Structure : StructureClass, DataType { };

interface UmlAssociationClassClass : ClassClass, AssociationClass {
   readonly attribute UmlAssociationClassUList
     all_of_kind_uml_association_class;
   readonly attribute UmlAssociationClassUList
     all_of_type_uml_association_class;
   UmlAssociationClass create_uml_association_class (in UmlCore::Name
     name,
                                                    in boolean is_root,
                                                    in boolean is_leaf,
                                                in boolean is_abstract,
                                                  in boolean is_active)
      raises (Reflective::SemanticError);
};

interface UmlAssociationClass : UmlAssociationClassClass,
                                Class, Association { };
```

```
interface ElementOwnershipClass : ElementClass {
  readonly attribute ElementOwnershipUList all_of_kind_element_ownership;
  readonly attribute ElementOwnershipUList all_of_type_element_ownership;
  ElementOwnership create_element_ownership (in VisibilityKind visibilty)
      raises (Reflective::SemanticError);
};

interface ElementOwnership : ElementOwnershipClass, Element {
   VisibilityKind visibilty ()
      raises (Reflective::SemanticError);
   void set_visibilty (in VisibilityKind new_value)
      raises (Reflective::SemanticError);
   UmlCore::Namespace namespace ()
      raises (Reflective::SemanticError);
   void set_namespace (in UmlCore::Namespace new_value)
      raises (Reflective::SemanticError);
   ModelElement owned_element ()
      raises (Reflective::SemanticError);
   void set_owned_element (in ModelElement new_value)
      raises (Reflective::SemanticError);
};

struct AssociationOwnsAssociationEndLink {
   UmlCore::Association association;
   AssociationEnd connection;
};
typedef sequence <AssociationOwnsAssociationEndLink>
  AssociationOwnsAssociationEndLinkSet;

interface AssociationOwnsAssociationEnd : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   AssociationOwnsAssociationEndLinkSet
     all_association_owns_association_end_links();
   boolean exists (in UmlCore::Association association,
                   in AssociationEnd connection);
   UmlCore::Association with_connection (in AssociationEnd connection);
   AssociationEndUList with_association (in UmlCore::Association
     association);
   void add (in UmlCore::Association association,
             in AssociationEnd connection)
      raises (Reflective::StructuralError, Reflective::SemanticError);
```

```
   void add_before_connection (in UmlCore::Association association,
                               in AssociationEnd connection,
                               in AssociationEnd before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_association (in UmlCore::Association association,
                            in AssociationEnd connection,
                            in UmlCore::Association new_association)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_connection (in UmlCore::Association association,
                           in AssociationEnd connection,
                           in AssociationEnd new_connection)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::Association association,
                in AssociationEnd connection)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ClassifierOwnsFeatureLink {
   Classifier owner;
   UmlCore::Feature feature;
};
typedef sequence <ClassifierOwnsFeatureLink>
     ClassifierOwnsFeatureLinkSet;

interface ClassifierOwnsFeature : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   ClassifierOwnsFeatureLinkSet all_classifier_owns_feature_links();
   boolean exists (in Classifier owner, in UmlCore::Feature feature);
   Classifier with_feature (in UmlCore::Feature feature);
   UmlCore::FeatureUList with_owner (in Classifier owner);
   void add (in Classifier owner, in UmlCore::Feature feature)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_feature (in Classifier owner,
```

```
                            in UmlCore::Feature feature,
                            in UmlCore::Feature before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_owner (in Classifier owner,
                      in UmlCore::Feature feature,
                      in Classifier new_owner)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_feature (in Classifier owner,
                        in UmlCore::Feature feature,
                        in UmlCore::Feature new_feature)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Classifier owner, in UmlCore::Feature feature)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct MethodIsSpecifiedByOperationLink {
   Operation specification;
   UmlCore::Method method;
};
typedef sequence <MethodIsSpecifiedByOperationLink>
  MethodIsSpecifiedByOperationLinkSet;

interface MethodIsSpecifiedByOperation : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   MethodIsSpecifiedByOperationLinkSet
     all_method_is_specified_by_operation_links();
  boolean exists (in Operation specification, in UmlCore::Method method);
   Operation with_method (in UmlCore::Method method);
   UmlCore::MethodSet with_specification (in Operation specification);
   void add (in Operation specification, in UmlCore::Method method)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_specification (in Operation specification,
                              in UmlCore::Method method,
```

```
                                  in Operation new_specification)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_method (in Operation specification,
                         in UmlCore::Method method,
                         in UmlCore::Method new_method)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void remove (in Operation specification, in UmlCore::Method method)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct StructuralFeatureIsOfTypeClassifierLink {
    StructuralFeature features;
    Classifier type;
};
typedef sequence <StructuralFeatureIsOfTypeClassifierLink>
  StructuralFeatureIsOfTypeClassifierLinkSet;

interface StructuralFeatureIsOfTypeClassifier :
      Reflective::RefAssociation {
    readonly attribute UmlCorePackage enclosing_package_ref;
    StructuralFeatureIsOfTypeClassifierLinkSet
      all_structural_feature_is_of_type_classifier_links();
    boolean exists (in StructuralFeature features, in Classifier type);
    StructuralFeatureUList with_type (in Classifier type);
    Classifier with_features (in StructuralFeature features);
    void add (in StructuralFeature features, in Classifier type)
        raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_before_features (in StructuralFeature features,
                               in Classifier type,
                               in StructuralFeature before)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_features (in StructuralFeature features,
                           in Classifier type,
```

```
                         in StructuralFeature new_features)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_type (in StructuralFeature features,
                     in Classifier type,
                     in Classifier new_type)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in StructuralFeature features, in Classifier type)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct NamespaceOwnsModelElementLink {
   UmlCore::Namespace namespace;
   ModelElement owned_element;
};
typedef sequence <NamespaceOwnsModelElementLink>
  NamespaceOwnsModelElementLinkSet;

interface NamespaceOwnsModelElement : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   NamespaceOwnsModelElementLinkSet
     all_namespace_owns_model_element_links();
   boolean exists (in UmlCore::Namespace namespace,
                   in ModelElement owned_element);
   UmlCore::Namespace with_owned_element (in ModelElement owned_element);
   ModelElementSet with_namespace (in UmlCore::Namespace namespace);
   void add (in UmlCore::Namespace namespace, in ModelElement
     owned_element)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_namespace (in UmlCore::Namespace namespace,
                          in ModelElement owned_element,
                          in UmlCore::Namespace new_namespace)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_owned_element (in UmlCore::Namespace namespace,
```

```
                              in ModelElement owned_element,
                              in ModelElement new_owned_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::Namespace namespace,
                in ModelElement owned_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct BehavioralFeatureOwnsParameterLink {
   UmlCore::BehavioralFeature behavioral_feature;
   UmlCore::Parameter parameter;
};
typedef sequence <BehavioralFeatureOwnsParameterLink>
  BehavioralFeatureOwnsParameterLinkSet;

interface BehavioralFeatureOwnsParameter : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   BehavioralFeatureOwnsParameterLinkSet
     all_behavioral_feature_owns_parameter_links();
   boolean exists (in UmlCore::BehavioralFeature behavioral_feature,
                   in UmlCore::Parameter parameter);
   UmlCore::BehavioralFeature with_parameter (
                                     in UmlCore::Parameter parameter);
   UmlCore::ParameterUList with_behavioral_feature (
                        in UmlCore::BehavioralFeature behavioral_feature);
   void add (in UmlCore::BehavioralFeature behavioral_feature,
             in UmlCore::Parameter parameter)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_parameter (
                     in UmlCore::BehavioralFeature behavioral_feature,
                          in UmlCore::Parameter parameter,
                          in UmlCore::Parameter before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_behavioral_feature (
                     in UmlCore::BehavioralFeature behavioral_feature,
```

```
                            in UmlCore::Parameter parameter,
                       in UmlCore::BehavioralFeature new_behavioral_feature)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_parameter (in UmlCore::BehavioralFeature
     behavioral_feature,
                             in UmlCore::Parameter parameter,
                             in UmlCore::Parameter new_parameter)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::BehavioralFeature behavioral_feature,
                in UmlCore::Parameter parameter)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ParameterIsOfTypeClassifierLink {
   Classifier type;
   UmlCore::Parameter parameter;
};
typedef sequence <ParameterIsOfTypeClassifierLink>
  ParameterIsOfTypeClassifierLinkSet;

interface ParameterIsOfTypeClassifier : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   ParameterIsOfTypeClassifierLinkSet
     all_parameter_is_of_type_classifier_links();
   boolean exists (in Classifier type, in UmlCore::Parameter parameter);
   Classifier with_parameter (in UmlCore::Parameter parameter);
   UmlCore::ParameterUList with_type (in Classifier type);
   void add (in Classifier type, in UmlCore::Parameter parameter)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_parameter (in Classifier type,
                                 in UmlCore::Parameter parameter,
                                 in UmlCore::Parameter before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
```

```
   void modify_type (in Classifier type,
                     in UmlCore::Parameter parameter,
                     in Classifier new_type)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_parameter (in Classifier type,
                          in UmlCore::Parameter parameter,
                          in UmlCore::Parameter new_parameter)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Classifier type, in UmlCore::Parameter parameter)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct GeneralizableElementIsSubtypeInGeneralizationLink {
   GeneralizableElement subtype;
   UmlCore::Generalization generalization;
};
typedef sequence <GeneralizableElementIsSubtypeInGeneralizationLink>
  GeneralizableElementIsSubtypeInGeneralizationLinkSet;

interface GeneralizableElementIsSubtypeInGeneralization :
                                           Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   GeneralizableElementIsSubtypeInGeneralizationLinkSet
     all_generalizable_element_is_subtype_in_generalization_links();
   boolean exists (in GeneralizableElement subtype,
                   in UmlCore::Generalization generalization);
   GeneralizableElement with_generalization (
                              in UmlCore::Generalization generalization);
   UmlCore::GeneralizationSet with_subtype (in GeneralizableElement
     subtype);
   void add (in GeneralizableElement subtype,
             in UmlCore::Generalization generalization)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_subtype (in GeneralizableElement subtype,
                        in UmlCore::Generalization generalization,
```

```
                           in GeneralizableElement new_subtype)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_generalization (in GeneralizableElement subtype,
                               in UmlCore::Generalization generalization,
                           in UmlCore::Generalization new_generalization)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in GeneralizableElement subtype,
                in UmlCore::Generalization generalization)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct GeneralizableElementIsSupertypeInGeneralizationLink {
   GeneralizableElement supertype;
   Generalization specialization;
};
typedef sequence <GeneralizableElementIsSupertypeInGeneralizationLink>
  GeneralizableElementIsSupertypeInGeneralizationLinkSet;

interface GeneralizableElementIsSupertypeInGeneralization :
                                             Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   GeneralizableElementIsSupertypeInGeneralizationLinkSet
     all_generalizable_element_is_supertype_in_generalization_links();
   boolean exists (in GeneralizableElement supertype,
                   in Generalization specialization);
   GeneralizableElement with_specialization (
                                      in Generalization specialization);
   GeneralizationSet with_supertype (in GeneralizableElement supertype);
   void add (in GeneralizableElement supertype,
             in Generalization specialization)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_supertype (in GeneralizableElement supertype,
                          in Generalization specialization,
                          in GeneralizableElement new_supertype)
      raises (Reflective::StructuralError,
```

```
                Reflective::SemanticError,
                Reflective::NotFound);
   void modify_specialization (in GeneralizableElement supertype,
                                 in Generalization specialization,
                                 in Generalization new_specialization)
      raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
   void remove (in GeneralizableElement supertype,
                  in Generalization specialization)
      raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct AssociationEndOwnsQualifierAttributeLink {
   UmlAttribute qualifier;
   UmlCore::AssociationEnd association_end;
};
typedef sequence <AssociationEndOwnsQualifierAttributeLink>
  AssociationEndOwnsQualifierAttributeLinkSet;

interface AssociationEndOwnsQualifierAttribute :
     Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   AssociationEndOwnsQualifierAttributeLinkSet
     all_association_end_owns_qualifier_attribute_links();
   boolean exists (in UmlAttribute qualifier,
                     in UmlCore::AssociationEnd association_end);
   UmlAttributeUList with_association_end (
                              in UmlCore::AssociationEnd association_end);
   UmlCore::AssociationEnd with_qualifier (in UmlAttribute qualifier);
   void add (in UmlAttribute qualifier,
               in UmlCore::AssociationEnd association_end)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_qualifier (in UmlAttribute qualifier,
                                in UmlCore::AssociationEnd association_end,
                                 in UmlAttribute before)
      raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
```

```
    void modify_qualifier (in UmlAttribute qualifier,
                           in UmlCore::AssociationEnd association_end,
                           in UmlAttribute new_qualifier)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_association_end (
                               in UmlAttribute qualifier,
                              in UmlCore::AssociationEnd association_end,
                          in UmlCore::AssociationEnd new_association_end)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in UmlAttribute qualifier,
                 in UmlCore::AssociationEnd association_end)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
 };

 struct AssociationEndIsOfTypeClassifierLink {
    Classifier type2;
    AssociationEnd participant;
 };
 typedef sequence <AssociationEndIsOfTypeClassifierLink>
   AssociationEndIsOfTypeClassifierLinkSet;

interface AssociationEndIsOfTypeClassifier : Reflective::RefAssociation {
    readonly attribute UmlCorePackage enclosing_package_ref;
    AssociationEndIsOfTypeClassifierLinkSet
      all_association_end_is_of_type_classifier_links();
    boolean exists (in Classifier type2, in AssociationEnd participant);
    Classifier with_participant (in AssociationEnd participant);
    AssociationEndSet with_type2 (in Classifier type2);
    void add (in Classifier type2, in AssociationEnd participant)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_type2 (in Classifier type2,
                       in AssociationEnd participant,
                       in Classifier new_type2)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
```

```
                 Reflective::NotFound);
      void modify_participant (in Classifier type2,
                               in AssociationEnd participant,
                               in AssociationEnd new_participant)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in Classifier type2, in AssociationEnd participant)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct ClassifierIsRealizedByClassifierLink {
      Classifier realization;
      Classifier specification;
   };
   typedef sequence <ClassifierIsRealizedByClassifierLink>
     ClassifierIsRealizedByClassifierLinkSet;

  interface ClassifierIsRealizedByClassifier : Reflective::RefAssociation {
      readonly attribute UmlCorePackage enclosing_package_ref;
      ClassifierIsRealizedByClassifierLinkSet
        all_classifier_is_realized_by_classifier_links();
     boolean exists (in Classifier realization, in Classifier specification);
      ClassifierSet with_specification (in Classifier specification);
      ClassifierSet with_realization (in Classifier realization);
      void add (in Classifier realization, in Classifier specification)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_realization (in Classifier realization,
                               in Classifier specification,
                               in Classifier new_realization)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_specification (in Classifier realization,
                                 in Classifier specification,
                                 in Classifier new_specification)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
```

```
   void remove (in Classifier realization, in Classifier specification)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct AssociationEndIsSpecifiedByClassifierLink {
   UmlCore::AssociationEnd association_end;
   Classifier specification;
};
typedef sequence <AssociationEndIsSpecifiedByClassifierLink>
  AssociationEndIsSpecifiedByClassifierLinkSet;

interface AssociationEndIsSpecifiedByClassifier :
                                           Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   AssociationEndIsSpecifiedByClassifierLinkSet
     all_association_end_is_specified_by_classifier_links();
   boolean exists (in UmlCore::AssociationEnd association_end,
                   in Classifier specification);
   UmlCore::AssociationEndSet with_specification (
                                           in Classifier specification);
   ClassifierSet with_association_end (
                            in UmlCore::AssociationEnd association_end);
   void add (in UmlCore::AssociationEnd association_end,
             in Classifier specification)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_association_end (
                             in UmlCore::AssociationEnd association_end,
                              in Classifier specification,
                         in UmlCore::AssociationEnd new_association_end)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
  void modify_specification (in UmlCore::AssociationEnd association_end,
                              in Classifier specification,
                              in Classifier new_specification)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::AssociationEnd association_end,
```

```
                   in Classifier specification)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct ModelElementIsSupplierInDependencyLink {
      ModelElement supplier;
      Dependency provision;
   };
   typedef sequence <ModelElementIsSupplierInDependencyLink>
     ModelElementIsSupplierInDependencyLinkSet;

  interface ModelElementIsSupplierInDependency : Reflective::RefAssociation
        {
      readonly attribute UmlCorePackage enclosing_package_ref;
      ModelElementIsSupplierInDependencyLinkSet
        all_model_element_is_supplier_in_dependency_links();
      boolean exists (in ModelElement supplier, in Dependency provision);
      ModelElementSet with_provision (in Dependency provision);
      DependencySet with_supplier (in ModelElement supplier);
      void add (in ModelElement supplier, in Dependency provision)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_supplier (in ModelElement supplier,
                            in Dependency provision,
                            in ModelElement new_supplier)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_provision (in ModelElement supplier,
                             in Dependency provision,
                             in Dependency new_provision)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in ModelElement supplier, in Dependency provision)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };
```

```
struct ModelElementOwnsTaggedValueLink {
   UmlCore::ModelElement model_element;
   UmlCore::TaggedValue tagged_value;
};
typedef sequence <ModelElementOwnsTaggedValueLink>
  ModelElementOwnsTaggedValueLinkSet;

interface ModelElementOwnsTaggedValue : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   ModelElementOwnsTaggedValueLinkSet
     all_model_element_owns_tagged_value_links();
   boolean exists (in UmlCore::ModelElement model_element,
                   in UmlCore::TaggedValue tagged_value);
   UmlCore::ModelElement with_tagged_value (
                                  in UmlCore::TaggedValue tagged_value);
   UmlCore::TaggedValueSet with_model_element (
                                in UmlCore::ModelElement model_element);
   void add (in UmlCore::ModelElement model_element,
             in UmlCore::TaggedValue tagged_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_model_element (in UmlCore::ModelElement model_element,
                              in UmlCore::TaggedValue tagged_value,
                             in UmlCore::ModelElement new_model_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_tagged_value (in UmlCore::ModelElement model_element,
                             in UmlCore::TaggedValue tagged_value,
                             in UmlCore::TaggedValue new_tagged_value)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::ModelElement model_element,
                in UmlCore::TaggedValue tagged_value)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ConstraintConstrainsModelElementLink {
   ModelElement constrained_element;
```

```
    UmlCore::Constraint constraint;
 };
 typedef sequence <ConstraintConstrainsModelElementLink>
   ConstraintConstrainsModelElementLinkSet;

interface ConstraintConstrainsModelElement : Reflective::RefAssociation {
    readonly attribute UmlCorePackage enclosing_package_ref;
    ConstraintConstrainsModelElementLinkSet
      all_constraint_constrains_model_element_links();
    boolean exists (in ModelElement constrained_element,
                    in UmlCore::Constraint constraint);
   ModelElementUList with_constraint (in UmlCore::Constraint constraint);
    UmlCore::ConstraintSet with_constrained_element (
                                    in ModelElement constrained_element);
    void add (in ModelElement constrained_element,
              in UmlCore::Constraint constraint)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_before_constrained_element (in ModelElement
      constrained_element,
                                        in UmlCore::Constraint constraint,
                                          in ModelElement before)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_constrained_element (in ModelElement constrained_element,
                                     in UmlCore::Constraint constraint,
                                 in ModelElement new_constrained_element)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_constraint (in ModelElement constrained_element,
                            in UmlCore::Constraint constraint,
                            in UmlCore::Constraint new_constraint)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in ModelElement constrained_element,
                 in UmlCore::Constraint constraint)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
```

```
};

struct ModelElementIsClientInDependencyLink {
   ModelElement client;
   Dependency requirement;
};
typedef sequence <ModelElementIsClientInDependencyLink>
  ModelElementIsClientInDependencyLinkSet;

interface ModelElementIsClientInDependency : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   ModelElementIsClientInDependencyLinkSet
     all_model_element_is_client_in_dependency_links();
   boolean exists (in ModelElement client, in Dependency requirement);
   ModelElementSet with_requirement (in Dependency requirement);
   DependencySet with_client (in ModelElement client);
   void add (in ModelElement client, in Dependency requirement)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_client (in ModelElement client,
                       in Dependency requirement,
                       in ModelElement new_client)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_requirement (in ModelElement client,
                            in Dependency requirement,
                            in Dependency new_requirement)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ModelElement client, in Dependency requirement)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct EnumerationOwnsEnumerationLiteralLink {
   UmlCore::Enumeration enumeration;
   EnumerationLiteral literal;
};
typedef sequence <EnumerationOwnsEnumerationLiteralLink>
```

```
   EnumerationOwnsEnumerationLiteralLinkSet;

interface EnumerationOwnsEnumerationLiteral : Reflective::RefAssociation {
    readonly attribute UmlCorePackage enclosing_package_ref;
    EnumerationOwnsEnumerationLiteralLinkSet
      all_enumeration_owns_enumeration_literal_links();
    boolean exists (in UmlCore::Enumeration enumeration,
                    in EnumerationLiteral literal);
    UmlCore::Enumeration with_literal (in EnumerationLiteral literal);
    EnumerationLiteralUList with_enumeration (
                                    in UmlCore::Enumeration enumeration);
    void add (in UmlCore::Enumeration enumeration,
              in EnumerationLiteral literal)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_before_literal (in UmlCore::Enumeration enumeration,
                             in EnumerationLiteral literal,
                             in EnumerationLiteral before)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_enumeration (in UmlCore::Enumeration enumeration,
                             in EnumerationLiteral literal,
                             in UmlCore::Enumeration new_enumeration)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_literal (in UmlCore::Enumeration enumeration,
                         in EnumerationLiteral literal,
                         in EnumerationLiteral new_literal)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in UmlCore::Enumeration enumeration,
                 in EnumerationLiteral literal)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
 };

 struct StereotypeOwnsRequiredTaggedValueLink {
    TaggedValue required_tag;
```

```
    UmlCore::Stereotype stereotype;
 };
 typedef sequence <StereotypeOwnsRequiredTaggedValueLink>
   StereotypeOwnsRequiredTaggedValueLinkSet;

interface StereotypeOwnsRequiredTaggedValue : Reflective::RefAssociation {
    readonly attribute UmlCorePackage enclosing_package_ref;
    StereotypeOwnsRequiredTaggedValueLinkSet
      all_stereotype_owns_required_tagged_value_links();
    boolean exists (in TaggedValue required_tag,
                    in UmlCore::Stereotype stereotype);
    TaggedValueSet with_stereotype (in UmlCore::Stereotype stereotype);
    UmlCore::Stereotype with_required_tag (in TaggedValue required_tag);
    void add (in TaggedValue required_tag, in UmlCore::Stereotype
      stereotype)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_required_tag (in TaggedValue required_tag,
                              in UmlCore::Stereotype stereotype,
                              in TaggedValue new_required_tag)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_stereotype (in TaggedValue required_tag,
                            in UmlCore::Stereotype stereotype,
                            in UmlCore::Stereotype new_stereotype)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in TaggedValue required_tag,
                 in UmlCore::Stereotype stereotype)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
 };

 struct StereotypeExtendsModelElementLink {
    UmlCore::Stereotype stereotype;
    ModelElement extended_element;
 };
 typedef sequence <StereotypeExtendsModelElementLink>
   StereotypeExtendsModelElementLinkSet;
```

```
interface StereotypeExtendsModelElement : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   StereotypeExtendsModelElementLinkSet
     all_stereotype_extends_model_element_links();
   boolean exists (in UmlCore::Stereotype stereotype,
                   in ModelElement extended_element);
   UmlCore::StereotypeSet with_extended_element (
                                     in ModelElement extended_element);
   ModelElementSet with_stereotype (in UmlCore::Stereotype stereotype);
   void add (in UmlCore::Stereotype stereotype,
             in ModelElement extended_element)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_stereotype (in UmlCore::Stereotype stereotype,
                           in ModelElement extended_element,
                           in UmlCore::Stereotype new_stereotype)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_extended_element (in UmlCore::Stereotype stereotype,
                                 in ModelElement extended_element,
                                 in ModelElement new_extended_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::Stereotype stereotype,
                in ModelElement extended_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ModelElementParemeterizedByModelElementLink {
   ModelElement template;
   ModelElement template_parameter;
};
typedef sequence <ModelElementParemeterizedByModelElementLink>
  ModelElementParemeterizedByModelElementLinkSet;

interface ModelElementParemeterizedByModelElement :
                                            Reflective::RefAssociation {
```

```
    readonly attribute UmlCorePackage enclosing_package_ref;
    ModelElementParemeterizedByModelElementLinkSet
      all_model_element_paremeterized_by_model_element_links();
    boolean exists (in ModelElement template,
                    in ModelElement template_parameter);
    ModelElement with_template_parameter (in ModelElement
      template_parameter);
    ModelElementUList with_template (in ModelElement template);
   void add (in ModelElement template, in ModelElement template_parameter)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_before_template_parameter (in ModelElement template,
                                        in ModelElement template_parameter,
                                          in ModelElement before)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_template (in ModelElement template,
                          in ModelElement template_parameter,
                          in ModelElement new_template)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_template_parameter (in ModelElement template,
                                    in ModelElement template_parameter,
                                  in ModelElement new_template_parameter)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in ModelElement template, in ModelElement
      template_parameter)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
};

struct NamespaceOwnsElementOwnershipLink {
    UmlCore::Namespace namespace;
    UmlCore::ElementOwnership element_ownership;
};
typedef sequence <NamespaceOwnsElementOwnershipLink>
  NamespaceOwnsElementOwnershipLinkSet;
```

```
interface NamespaceOwnsElementOwnership : Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   NamespaceOwnsElementOwnershipLinkSet
     all_namespace_owns_element_ownership_links();
   boolean exists (in UmlCore::Namespace namespace,
                   in UmlCore::ElementOwnership element_ownership);
   UmlCore::Namespace with_element_ownership (
                        in UmlCore::ElementOwnership element_ownership);
   UmlCore::ElementOwnershipSet with_namespace (
                                       in UmlCore::Namespace namespace);
   void add (in UmlCore::Namespace namespace,
             in UmlCore::ElementOwnership element_ownership)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_namespace (in UmlCore::Namespace namespace,
                          in UmlCore::ElementOwnership element_ownership,
                           in UmlCore::Namespace new_namespace)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_element_ownership (
                           in UmlCore::Namespace namespace,
                          in UmlCore::ElementOwnership element_ownership,
                      in UmlCore::ElementOwnership new_element_ownership)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCore::Namespace namespace,
                in UmlCore::ElementOwnership element_ownership)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ModelElementIsOwnedViaElementOwnershipLink {
   ModelElement owned_element;
   ElementOwnership namespace1;
};
typedef sequence <ModelElementIsOwnedViaElementOwnershipLink>
  ModelElementIsOwnedViaElementOwnershipLinkSet;
```

```
interface ModelElementIsOwnedViaElementOwnership :
                                                    Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   ModelElementIsOwnedViaElementOwnershipLinkSet
     all_model_element_is_owned_via_element_ownership_links();
   boolean exists (in ModelElement owned_element,
                   in ElementOwnership namespace1);
   ModelElement with_namespace1 (in ElementOwnership namespace1);
   ElementOwnership with_owned_element (in ModelElement owned_element);
  void add (in ModelElement owned_element, in ElementOwnership namespace1)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_owned_element (in ModelElement owned_element,
                              in ElementOwnership namespace1,
                              in ModelElement new_owned_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_namespace1 (in ModelElement owned_element,
                           in ElementOwnership namespace1,
                           in ElementOwnership new_namespace1)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ModelElement owned_element, in ElementOwnership
     namespace1)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct StereotypeIsConstrainedByConstraintLink {
   Constraint stereotype_constraint;
   Stereotype constrained_stereotype;
};
typedef sequence <StereotypeIsConstrainedByConstraintLink>
  StereotypeIsConstrainedByConstraintLinkSet;

interface StereotypeIsConstrainedByConstraint :
     Reflective::RefAssociation {
   readonly attribute UmlCorePackage enclosing_package_ref;
   StereotypeIsConstrainedByConstraintLinkSet
```

```
        all_stereotype_is_constrained_by_constraint_links();
      boolean exists (in Constraint stereotype_constraint,
                      in Stereotype constrained_stereotype);
      ConstraintSet with_constrained_stereotype (
                                     in Stereotype constrained_stereotype);
      Stereotype with_stereotype_constraint (
                                     in Constraint stereotype_constraint);
      void add (in Constraint stereotype_constraint,
                in Stereotype constrained_stereotype)
         raises (Reflective::StructuralError, Reflective::SemanticError);
     void modify_stereotype_constraint (in Constraint stereotype_constraint,
                                     in Stereotype constrained_stereotype,
                                   in Constraint new_stereotype_constraint)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_constrained_stereotype (
                                     in Constraint stereotype_constraint,
                                     in Stereotype constrained_stereotype,
                                   in Stereotype new_constrained_stereotype)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in Constraint stereotype_constraint,
                   in Stereotype constrained_stereotype)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   interface UmlCorePackageFactory  {
      UmlCorePackage create_uml_core_package ()
         raises (Reflective::SemanticError);
   };

   interface UmlCorePackage : Reflective::RefPackage {
      readonly attribute ElementClass element_class_ref;
      readonly attribute TaggedValueClass tagged_value_class_ref;
      readonly attribute EnumerationLiteralClass
        enumeration_literal_class_ref;
      readonly attribute ModelElementClass model_element_class_ref;
```

```
    readonly attribute FeatureClass feature_class_ref;
    readonly attribute GeneralizationClass generalization_class_ref;
    readonly attribute NamespaceClass namespace_class_ref;
    readonly attribute ParameterClass parameter_class_ref;
    readonly attribute ConstraintClass constraint_class_ref;
    readonly attribute DependencyClass dependency_class_ref;
    readonly attribute RequestClass request_class_ref;
    readonly attribute GeneralizableElementClass
      generalizable_element_class_ref;
   readonly attribute BehavioralFeatureClass behavioral_feature_class_ref;
    readonly attribute ClassifierClass classifier_class_ref;
    readonly attribute OperationClass operation_class_ref;
    readonly attribute StereotypeClass stereotype_class_ref;
   readonly attribute StructuralFeatureClass structural_feature_class_ref;
    readonly attribute DataTypeClass data_type_class_ref;
    readonly attribute UmlInterfaceClass uml_interface_class_ref;
    readonly attribute UmlAttributeClass uml_attribute_class_ref;
    readonly attribute AssociationEndClass association_end_class_ref;
    readonly attribute AssociationClass association_class_ref;
    readonly attribute MethodClass method_class_ref;
    readonly attribute EnumerationClass enumeration_class_ref;
    readonly attribute ClassClass class_class_ref;
    readonly attribute PrimitiveClass primitive_class_ref;
    readonly attribute StructureClass structure_class_ref;
    readonly attribute UmlAssociationClassClass
      uml_association_class_class_ref;
    readonly attribute ElementOwnershipClass element_ownership_class_ref;

    readonly attribute AssociationOwnsAssociationEnd
      association_owns_association_end_ref;
    readonly attribute ClassifierOwnsFeature classifier_owns_feature_ref;
    readonly attribute MethodIsSpecifiedByOperation
      method_is_specified_by_operation_ref;
    readonly attribute StructuralFeatureIsOfTypeClassifier
      structural_feature_is_of_type_classifier_ref;
    readonly attribute NamespaceOwnsModelElement
      namespace_owns_model_element_ref;
    readonly attribute BehavioralFeatureOwnsParameter
      behavioral_feature_owns_parameter_ref;
    readonly attribute ParameterIsOfTypeClassifier
      parameter_is_of_type_classifier_ref;
```

```
        readonly attribute GeneralizableElementIsSubtypeInGeneralization
          generalizable_element_is_subtype_in_generalization_ref;
        readonly attribute GeneralizableElementIsSupertypeInGeneralization
          generalizable_element_is_supertype_in_generalization_ref;
        readonly attribute AssociationEndOwnsQualifierAttribute
          association_end_owns_qualifier_attribute_ref;
        readonly attribute AssociationEndIsOfTypeClassifier
          association_end_is_of_type_classifier_ref;
        readonly attribute ClassifierIsRealizedByClassifier
          classifier_is_realized_by_classifier_ref;
        readonly attribute AssociationEndIsSpecifiedByClassifier
          association_end_is_specified_by_classifier_ref;
        readonly attribute ModelElementIsSupplierInDependency
          model_element_is_supplier_in_dependency_ref;
        readonly attribute ModelElementOwnsTaggedValue
          model_element_owns_tagged_value_ref;
        readonly attribute ConstraintConstrainsModelElement
          constraint_constrains_model_element_ref;
        readonly attribute ModelElementIsClientInDependency
          model_element_is_client_in_dependency_ref;
        readonly attribute EnumerationOwnsEnumerationLiteral
          enumeration_owns_enumeration_literal_ref;
        readonly attribute StereotypeOwnsRequiredTaggedValue
          stereotype_owns_required_tagged_value_ref;
        readonly attribute StereotypeExtendsModelElement
          stereotype_extends_model_element_ref;
        readonly attribute ModelElementParemeterizedByModelElement
          model_element_paremeterized_by_model_element_ref;
        readonly attribute NamespaceOwnsElementOwnership
          namespace_owns_element_ownership_ref;
        readonly attribute ModelElementIsOwnedViaElementOwnership
          model_element_is_owned_via_element_ownership_ref;
        readonly attribute StereotypeIsConstrainedByConstraint
          stereotype_is_constrained_by_constraint_ref;
    };
};
```

5.4.2 UMLModelManagement

```
#include "UmlCore.idl"

module UmlModelManagement {
   interface UmlModelManagementPackage;
   interface ElementReference;
   interface ElementReferenceClass;
   typedef sequence<ElementReference> ElementReferenceUList;
   typedef sequence<ElementReference> ElementReferenceSet;
   interface Model;
   interface ModelClass;
   typedef sequence<Model> ModelUList;
   interface Package;
   interface PackageClass;
   typedef sequence<Package> PackageUList;
   typedef sequence<Package> PackageSet;
   interface Subsystem;
   interface SubsystemClass;
   typedef sequence<Subsystem> SubsystemUList;

   interface PackageClass : ::UmlCore::GeneralizableElementClass {
      readonly attribute PackageUList all_of_kind_package;
      readonly attribute PackageUList all_of_type_package;
      Package create_package (in ::UmlCore::Name name,
                              in boolean is_root,
                              in boolean is_leaf,
                              in boolean is_abstract)
         raises (Reflective::SemanticError);
   };

   interface Package : PackageClass, ::UmlCore::GeneralizableElement {
      ElementReferenceSet supplier_element_reference ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_supplier_element_reference (in ElementReferenceSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_supplier_element_reference ()
         raises (Reflective::SemanticError);
      ::UmlCore::ModelElementSet referenced_element ()
         raises (Reflective::SemanticError);
      void add_referenced_element (in ::UmlCore::ModelElementSet new_value)
```

```
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_referenced_element ()
      raises (Reflective::SemanticError);
};

interface ModelClass : PackageClass {
   readonly attribute ModelUList all_of_kind_model;
   readonly attribute ModelUList all_of_type_model;
   Model create_model (in ::UmlCore::Name name,
                       in boolean is_root,
                       in boolean is_leaf,
                       in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Model : ModelClass, Package { };

interface SubsystemClass : PackageClass, ::UmlCore::ClassifierClass {
   readonly attribute SubsystemUList all_of_kind_subsystem;
   readonly attribute SubsystemUList all_of_type_subsystem;
   Subsystem create_subsystem (in ::UmlCore::Name name,
                               in boolean is_root,
                               in boolean is_leaf,
                               in boolean is_abstract,
                               in boolean is_instantiable)
      raises (Reflective::SemanticError);
};

interface Subsystem : SubsystemClass, Package, ::UmlCore::Classifier {
   boolean is_instantiable ()
      raises (Reflective::SemanticError);
   void set_is_instantiable (in boolean new_value)
      raises (Reflective::SemanticError);
};

interface ElementReferenceClass : ::UmlCore::ElementClass {
   readonly attribute ElementReferenceUList all_of_kind_element_reference;
   readonly attribute ElementReferenceUList all_of_type_element_reference;
   ElementReference create_element_reference (
                                 in ::UmlCore::VisibilityKind visibility,
                                    in ::UmlCore::Name alias)
```

```
      raises (Reflective::SemanticError);
};

interface ElementReference : ElementReferenceClass, ::UmlCore::Element {
   ::UmlCore::VisibilityKind visibility ()
      raises (Reflective::SemanticError);
   void set_visibility (in ::UmlCore::VisibilityKind new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::Name alias ()
      raises (Reflective::SemanticError);
   void set_alias (in ::UmlCore::Name new_value)
      raises (Reflective::SemanticError);
   Package referencing_package ()
      raises (Reflective::SemanticError);
   void set_referencing_package (in Package new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::ModelElementSet referenced_element ()
      raises (Reflective::SemanticError);
   void add_referenced_element (in ::UmlCore::ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_referenced_element ()
      raises (Reflective::SemanticError);
};

struct PackageElementReferenceLink {
   Package referencing_package;
   ElementReference supplier_element_reference;
};
typedef sequence <PackageElementReferenceLink>
      PackageElementReferenceLinkSet;

interface PackageElementReference : Reflective::RefAssociation {
   readonly attribute UmlModelManagementPackage enclosing_package_ref;
   PackageElementReferenceLinkSet all_package_element_reference_links();
   boolean exists (in Package referencing_package,
                   in ElementReference supplier_element_reference);
   Package with_supplier_element_reference (
                         in ElementReference supplier_element_reference);
   ElementReferenceSet with_referencing_package (
                                         in Package referencing_package);
   void add (in Package referencing_package,
```

```
             in ElementReference supplier_element_reference)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_referencing_package (
                              in Package referencing_package,
                          in ElementReference supplier_element_reference,
                              in Package new_referencing_package)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_supplier_element_reference (
                          in Package referencing_package,
                          in ElementReference supplier_element_reference,
                      in ElementReference new_supplier_element_reference)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Package referencing_package,
                in ElementReference supplier_element_reference)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ElementReferenceReferencesModelElementLink {
   ElementReference client_element_reference;
   ::UmlCore::ModelElement referenced_element;
};
typedef sequence <ElementReferenceReferencesModelElementLink>
  ElementReferenceReferencesModelElementLinkSet;

interface ElementReferenceReferencesModelElement :
                                            Reflective::RefAssociation {
   readonly attribute UmlModelManagementPackage enclosing_package_ref;
   ElementReferenceReferencesModelElementLinkSet
     all_element_reference_references_model_element_links();
   boolean exists (in ElementReference client_element_reference,
                   in ::UmlCore::ModelElement referenced_element);
   ElementReferenceSet with_referenced_element (
                          in ::UmlCore::ModelElement referenced_element);
   ::UmlCore::ModelElementSet with_client_element_reference (
                           in ElementReference client_element_reference);
```

```
   void add (in ElementReference client_element_reference,
             in ::UmlCore::ModelElement referenced_element)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_client_element_reference (
                            in ElementReference client_element_reference,
                           in ::UmlCore::ModelElement referenced_element,
                        in ElementReference new_client_element_reference)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_referenced_element (
                           in ElementReference client_element_reference,
                           in ::UmlCore::ModelElement referenced_element,
                       in ::UmlCore::ModelElement new_referenced_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ElementReference client_element_reference,
                in ::UmlCore::ModelElement referenced_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct PackageReferencesModelElementLink {
   ::UmlCore::ModelElement referenced_element;
   Package referencing_package;
};
typedef sequence <PackageReferencesModelElementLink>
  PackageReferencesModelElementLinkSet;

interface PackageReferencesModelElement : Reflective::RefAssociation {
   readonly attribute UmlModelManagementPackage enclosing_package_ref;
   PackageReferencesModelElementLinkSet
     all_package_references_model_element_links();
   boolean exists (in ::UmlCore::ModelElement referenced_element,
                   in Package referencing_package);
   ::UmlCore::ModelElementSet with_referencing_package (
                                        in Package referencing_package);
   PackageSet with_referenced_element (
                          in ::UmlCore::ModelElement referenced_element);
```

```
   void add (in ::UmlCore::ModelElement referenced_element,
             in Package referencing_package)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_referenced_element (
                          in ::UmlCore::ModelElement referenced_element,
                          in Package referencing_package,
                      in ::UmlCore::ModelElement new_referenced_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_referencing_package (
                          in ::UmlCore::ModelElement referenced_element,
                              in Package referencing_package,
                              in Package new_referencing_package)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCore::ModelElement referenced_element,
                in Package referencing_package)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

interface UmlModelManagementPackageFactory {
   UmlModelManagementPackage create_uml_model_management_package ()
      raises (Reflective::SemanticError);
};

interface UmlModelManagementPackage : Reflective::RefPackage {
   readonly attribute PackageClass package_class_ref;
   readonly attribute ModelClass model_class_ref;
   readonly attribute SubsystemClass subsystem_class_ref;
   readonly attribute ElementReferenceClass element_reference_class_ref;

   readonly attribute PackageElementReference
      package_element_reference_ref;
   readonly attribute ElementReferenceReferencesModelElement
     element_reference_references_model_element_ref;
   readonly attribute PackageReferencesModelElement
     package_references_model_element_ref;
```

```
    };
};
```

5.4.3 UMLAuxiliaryElements

```
#include "UmlCore.idl"

module UmlAuxiliaryElements {
    interface UmlAuxiliaryElementsPackage;
    interface Usage;
    interface UsageClass;
    typedef sequence<Usage> UsageUList;
    interface Component;
    interface ComponentClass;
    typedef sequence<Component> ComponentUList;
    typedef sequence<Component> ComponentSet;
    interface Presentation;
    interface PresentationClass;
    typedef sequence<Presentation> PresentationUList;
    typedef sequence<Presentation> PresentationSet;
    interface ViewElement;
    interface ViewElementClass;
    typedef sequence<ViewElement> ViewElementUList;
    typedef sequence<ViewElement> ViewElementSet;
    interface Binding;
    interface BindingClass;
    typedef sequence<Binding> BindingUList;
    interface Comment;
    interface CommentClass;
    typedef sequence<Comment> CommentUList;
    interface Trace;
    interface TraceClass;
    typedef sequence<Trace> TraceUList;
    interface Node;
    interface NodeClass;
    typedef sequence<Node> NodeUList;
    typedef sequence<Node> NodeSet;
    interface Refinement;
    interface RefinementClass;
    typedef sequence<Refinement> RefinementUList;

```

```
interface ComponentClass : ::UmlCore::ClassifierClass {
   readonly attribute ComponentUList all_of_kind_component;
   readonly attribute ComponentUList all_of_type_component;
   Component create_component (in ::UmlCore::Name name,
                               in boolean is_root,
                               in boolean is_leaf,
                               in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Component : ComponentClass, ::UmlCore::Classifier {
   ::UmlCore::ModelElementSet model_element ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_model_element (in ::UmlCore::ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_model_element ()
      raises (Reflective::SemanticError);
   NodeSet deployment ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_deployment (in NodeSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_deployment ()
      raises (Reflective::SemanticError);
};

interface NodeClass : ::UmlCore::ClassifierClass {
   readonly attribute NodeUList all_of_kind_node;
   readonly attribute NodeUList all_of_type_node;
   Node create_node (in ::UmlCore::Name name,
                     in boolean is_root,
                     in boolean is_leaf,
                     in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface Node : NodeClass, ::UmlCore::Classifier {
   UmlAuxiliaryElements::ComponentSet component ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_component (in UmlAuxiliaryElements::ComponentSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_component ()
```

```
        raises (Reflective::SemanticError);
  };

  interface PresentationClass : ::UmlCore::ElementClass {
     readonly attribute PresentationUList all_of_kind_presentation;
     readonly attribute PresentationUList all_of_type_presentation;
     Presentation create_presentation (in ::UmlCore::Geometry geometry,
                                       in ::UmlCore::GraphicMarker style)
        raises (Reflective::SemanticError);
  };

  interface Presentation : PresentationClass, ::UmlCore::Element {
     ::UmlCore::Geometry geometry ()
        raises (Reflective::SemanticError);
     void set_geometry (in ::UmlCore::Geometry new_value)
        raises (Reflective::SemanticError);
     ::UmlCore::GraphicMarker style ()
        raises (Reflective::SemanticError);
     void set_style (in ::UmlCore::GraphicMarker new_value)
        raises (Reflective::SemanticError);
     ::UmlCore::ModelElement model ()
        raises (Reflective::SemanticError);
     void set_model (in ::UmlCore::ModelElement new_value)
        raises (Reflective::SemanticError);
     ViewElement view ()
        raises (Reflective::SemanticError);
     void set_view (in ViewElement new_value)
        raises (Reflective::SemanticError);
  };

  interface ViewElementClass : ::UmlCore::ElementClass {
     readonly attribute ViewElementUList all_of_kind_view_element;
  };

  interface ViewElement : ViewElementClass, ::UmlCore::Element {
     UmlAuxiliaryElements::PresentationSet presentation ()
        raises (Reflective::NotSet, Reflective::SemanticError);
     void add_presentation (in UmlAuxiliaryElements::PresentationSet
        new_value)
        raises (Reflective::StructuralError, Reflective::SemanticError);
     void remove_presentation ()
```

```
      raises (Reflective::SemanticError);
   ::UmlCore::ModelElementSet model ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_model (in ::UmlCore::ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_model ()
      raises (Reflective::SemanticError);
};

interface BindingClass : ::UmlCore::DependencyClass {
   readonly attribute BindingUList all_of_kind_binding;
   readonly attribute BindingUList all_of_type_binding;
   Binding create_binding (in ::UmlCore::Name name,
                           in string description)
      raises (Reflective::SemanticError);
};

interface Binding : BindingClass, ::UmlCore::Dependency {
   ::UmlCore::ModelElementSet argument ()
      raises (Reflective::SemanticError);
   void add_argument (in ::UmlCore::ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_argument ()
      raises (Reflective::SemanticError);
};

interface CommentClass : ::UmlCore::ModelElementClass {
   readonly attribute CommentUList all_of_kind_comment;
   readonly attribute CommentUList all_of_type_comment;
   Comment create_comment (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Comment : CommentClass, ::UmlCore::ModelElement { };

interface RefinementClass : ::UmlCore::DependencyClass {
   readonly attribute RefinementUList all_of_kind_refinement;
   readonly attribute RefinementUList all_of_type_refinement;
   Refinement create_refinement (in ::UmlCore::Name name,
                                 in string description,
                                 in ::UmlCore::Mapping mapping)
```

```
      raises (Reflective::SemanticError);
};

interface Refinement : RefinementClass, ::UmlCore::Dependency {
   ::UmlCore::Mapping mapping ()
      raises (Reflective::SemanticError);
   void set_mapping (in ::UmlCore::Mapping new_value)
      raises (Reflective::SemanticError);
};

interface TraceClass : ::UmlCore::DependencyClass {
   readonly attribute TraceUList all_of_kind_trace;
   readonly attribute TraceUList all_of_type_trace;
   Trace create_trace (in ::UmlCore::Name name,
                       in string description)
      raises (Reflective::SemanticError);
};

interface Trace : TraceClass, ::UmlCore::Dependency { };

interface UsageClass : ::UmlCore::DependencyClass {
   readonly attribute UsageUList all_of_kind_usage;
   readonly attribute UsageUList all_of_type_usage;
   Usage create_usage (in ::UmlCore::Name name,
                       in string description)
      raises (Reflective::SemanticError);
};

interface Usage : UsageClass, ::UmlCore::Dependency { };

struct ComponentImplementsModelElementLink {
   Component implementation;
   ::UmlCore::ModelElement model_element;
};
typedef sequence <ComponentImplementsModelElementLink>
  ComponentImplementsModelElementLinkSet;

interface ComponentImplementsModelElement : Reflective::RefAssociation {
   readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
   ComponentImplementsModelElementLinkSet
     all_component_implements_model_element_links();
```

```
   boolean exists (in Component implementation,
                   in ::UmlCore::ModelElement model_element);
   ComponentSet with_model_element (in ::UmlCore::ModelElement
      model_element);
   ::UmlCore::ModelElementSet with_implementation (
                                            in Component implementation);
   void add (in Component implementation,
             in ::UmlCore::ModelElement model_element)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_implementation (in Component implementation,
                               in ::UmlCore::ModelElement model_element,
                               in Component new_implementation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_model_element (in Component implementation,
                              in ::UmlCore::ModelElement model_element,
                            in ::UmlCore::ModelElement new_model_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Component implementation,
                in ::UmlCore::ModelElement model_element)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct NodeDeploysComponentLink {
   Node deployment;
   UmlAuxiliaryElements::Component component;
};
typedef sequence <NodeDeploysComponentLink> NodeDeploysComponentLinkSet;

interface NodeDeploysComponent : Reflective::RefAssociation {
   readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
   NodeDeploysComponentLinkSet all_node_deploys_component_links();
   boolean exists (in Node deployment,
                   in UmlAuxiliaryElements::Component component);
   NodeSet with_component (in UmlAuxiliaryElements::Component component);
  UmlAuxiliaryElements::ComponentSet with_deployment (in Node deployment);
```

```
   void add (in Node deployment, in UmlAuxiliaryElements::Component
      component)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_deployment (in Node deployment,
                           in UmlAuxiliaryElements::Component component,
                           in Node new_deployment)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_component (in Node deployment,
                          in UmlAuxiliaryElements::Component component,
                        in UmlAuxiliaryElements::Component new_component)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Node deployment,
                in UmlAuxiliaryElements::Component component)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct PresentationPresentsModelElementLink {
   ::UmlCore::ModelElement model;
   UmlAuxiliaryElements::Presentation presentation;
};
typedef sequence <PresentationPresentsModelElementLink>
  PresentationPresentsModelElementLinkSet;

interface PresentationPresentsModelElement : Reflective::RefAssociation {
   readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
   PresentationPresentsModelElementLinkSet
     all_presentation_presents_model_element_links();
   boolean exists (in ::UmlCore::ModelElement model,
                   in UmlAuxiliaryElements::Presentation presentation);
   ::UmlCore::ModelElement with_presentation (
                     in UmlAuxiliaryElements::Presentation presentation);
   UmlAuxiliaryElements::PresentationSet with_model (
                                      in ::UmlCore::ModelElement model);
   void add (in ::UmlCore::ModelElement model,
             in UmlAuxiliaryElements::Presentation presentation)
```

```
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_model (in ::UmlCore::ModelElement model,
                      in UmlAuxiliaryElements::Presentation presentation,
                      in ::UmlCore::ModelElement new_model)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_presentation (
                      in ::UmlCore::ModelElement model,
                      in UmlAuxiliaryElements::Presentation presentation,
                  in UmlAuxiliaryElements::Presentation new_presentation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCore::ModelElement model,
                in UmlAuxiliaryElements::Presentation presentation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct PresentationPresentsViewElementLink {
   ViewElement view;
   UmlAuxiliaryElements::Presentation presentation;
};
typedef sequence <PresentationPresentsViewElementLink>
  PresentationPresentsViewElementLinkSet;

interface PresentationPresentsViewElement : Reflective::RefAssociation {
   readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
   PresentationPresentsViewElementLinkSet
     all_presentation_presents_view_element_links();
   boolean exists (in ViewElement view,
                   in UmlAuxiliaryElements::Presentation presentation);
   ViewElement with_presentation (
                     in UmlAuxiliaryElements::Presentation presentation);
   UmlAuxiliaryElements::PresentationSet with_view (in ViewElement view);
   void add (in ViewElement view,
             in UmlAuxiliaryElements::Presentation presentation)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_view (in ViewElement view,
```

```
                     in UmlAuxiliaryElements::Presentation presentation,
                     in ViewElement new_view)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_presentation (
                      in ViewElement view,
                      in UmlAuxiliaryElements::Presentation presentation,
                  in UmlAuxiliaryElements::Presentation new_presentation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ViewElement view,
                in UmlAuxiliaryElements::Presentation presentation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct BindingOwnsArgumentModelElementLink {
   UmlAuxiliaryElements::Binding binding;
   ::UmlCore::ModelElement argument;
};
typedef sequence <BindingOwnsArgumentModelElementLink>
  BindingOwnsArgumentModelElementLinkSet;

interface BindingOwnsArgumentModelElement : Reflective::RefAssociation {
   readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
   BindingOwnsArgumentModelElementLinkSet
     all_binding_owns_argument_model_element_links();
   boolean exists (in UmlAuxiliaryElements::Binding binding,
                   in ::UmlCore::ModelElement argument);
   UmlAuxiliaryElements::Binding with_argument (
                                    in ::UmlCore::ModelElement argument);
   ::UmlCore::ModelElementSet with_binding (
                               in UmlAuxiliaryElements::Binding binding);
   void add (in UmlAuxiliaryElements::Binding binding,
             in ::UmlCore::ModelElement argument)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_binding (in UmlAuxiliaryElements::Binding binding,
                        in ::UmlCore::ModelElement argument,
```

```
                            in UmlAuxiliaryElements::Binding new_binding)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_argument (in UmlAuxiliaryElements::Binding binding,
                          in ::UmlCore::ModelElement argument,
                          in ::UmlCore::ModelElement new_argument)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in UmlAuxiliaryElements::Binding binding,
                 in ::UmlCore::ModelElement argument)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
};

struct ViewElementProvidesViewOfModelElementLink {
    ViewElement view;
    ::UmlCore::ModelElement model;
};
typedef sequence <ViewElementProvidesViewOfModelElementLink>
  ViewElementProvidesViewOfModelElementLinkSet;

interface ViewElementProvidesViewOfModelElement :
                                            Reflective::RefAssociation {
    readonly attribute UmlAuxiliaryElementsPackage enclosing_package_ref;
    ViewElementProvidesViewOfModelElementLinkSet
      all_view_element_provides_view_of_model_element_links();
   boolean exists (in ViewElement view, in ::UmlCore::ModelElement model);
    ViewElementSet with_model (in ::UmlCore::ModelElement model);
    ::UmlCore::ModelElementSet with_view (in ViewElement view);
    void add (in ViewElement view, in ::UmlCore::ModelElement model)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_view (in ViewElement view,
                      in ::UmlCore::ModelElement model,
                      in ViewElement new_view)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_model (in ViewElement view,
```

```
                           in ::UmlCore::ModelElement model,
                           in ::UmlCore::ModelElement new_model)
            raises (Reflective::StructuralError,
                    Reflective::SemanticError,
                    Reflective::NotFound);
         void remove (in ViewElement view, in ::UmlCore::ModelElement model)
            raises (Reflective::StructuralError,
                    Reflective::SemanticError,
                    Reflective::NotFound);
      };

      interface UmlAuxiliaryElementsPackageFactory  {
         UmlAuxiliaryElementsPackage create_uml_auxiliary_elements_package ()
            raises (Reflective::SemanticError);
      };

      interface UmlAuxiliaryElementsPackage : Reflective::RefPackage {
         readonly attribute ComponentClass component_class_ref;
         readonly attribute NodeClass node_class_ref;
         readonly attribute PresentationClass presentation_class_ref;
         readonly attribute ViewElementClass view_element_class_ref;
         readonly attribute BindingClass binding_class_ref;
         readonly attribute CommentClass comment_class_ref;
         readonly attribute RefinementClass refinement_class_ref;
         readonly attribute TraceClass trace_class_ref;
         readonly attribute UsageClass usage_class_ref;

         readonly attribute ComponentImplementsModelElement
           component_implements_model_element_ref;
         readonly attribute NodeDeploysComponent node_deploys_component_ref;
         readonly attribute PresentationPresentsModelElement
           presentation_presents_model_element_ref;
         readonly attribute PresentationPresentsViewElement
           presentation_presents_view_element_ref;
         readonly attribute BindingOwnsArgumentModelElement
           binding_owns_argument_model_element_ref;
         readonly attribute ViewElementProvidesViewOfModelElement
           view_element_provides_view_of_model_element_ref;
      };
   };
```

5.4.4 UMLCollaborations

```
#include "UmlCommonBehavior.idl"

module UmlCollaborations {
   interface UmlCollaborationsPackage;
   interface Message;
   interface MessageClass;
   typedef sequence<Message> MessageUList;
   typedef sequence<Message> MessageSet;
   interface ClassifierRole;
   interface ClassifierRoleClass;
   typedef sequence<ClassifierRole> ClassifierRoleUList;
   typedef sequence<ClassifierRole> ClassifierRoleSet;
   interface AssociationRole;
   interface AssociationRoleClass;
   typedef sequence<AssociationRole> AssociationRoleUList;
   typedef sequence<AssociationRole> AssociationRoleSet;
   interface Interaction;
   interface InteractionClass;
   typedef sequence<Interaction> InteractionUList;
   interface AssociationEndRole;
   interface AssociationEndRoleClass;
   typedef sequence<AssociationEndRole> AssociationEndRoleUList;
   typedef sequence<AssociationEndRole> AssociationEndRoleSet;
   interface Collaboration;
   interface CollaborationClass;
   typedef sequence<Collaboration> CollaborationUList;
   typedef sequence<Collaboration> CollaborationSet;

   interface AssociationEndRoleClass : ::UmlCore::AssociationEndClass {
      readonly attribute AssociationEndRoleUList
        all_of_kind_association_end_role;
      readonly attribute AssociationEndRoleUList
        all_of_type_association_end_role;
      AssociationEndRole create_association_end_role (
                                      in ::UmlCore::Name name,
                                      in boolean is_navigable,
                                      in boolean is_ordered,
                                  in ::UmlCore::AggregationKind aggregation,
                                      in ::UmlCore::ScopeKind target_scope,
```

```
                                    in ::UmlCore::Multiplicity multiplicity,
                                    in ::UmlCore::ChangeableKind changeable)
      raises (Reflective::SemanticError);
};

interface AssociationEndRole : AssociationEndRoleClass,
                               ::UmlCore::AssociationEnd {
   ::UmlCore::AssociationEnd base ()
      raises (Reflective::SemanticError);
   void set_base (in ::UmlCore::AssociationEnd new_value)
      raises (Reflective::SemanticError);
};

interface ClassifierRoleClass : ::UmlCore::ClassifierClass {
   readonly attribute ClassifierRoleUList all_of_kind_classifier_role;
   readonly attribute ClassifierRoleUList all_of_type_classifier_role;
   ClassifierRole create_classifier_role (
                                    in ::UmlCore::Name name,
                                    in boolean is_root,
                                    in boolean is_leaf,
                                    in boolean is_abstract,
                                    in ::UmlCore::Multiplicity multiplicity)
      raises (Reflective::SemanticError);
};

interface ClassifierRole : ClassifierRoleClass, ::UmlCore::Classifier {
   ::UmlCore::Multiplicity multiplicity ()
      raises (Reflective::SemanticError);
   void set_multiplicity (in ::UmlCore::Multiplicity new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::Classifier base ()
      raises (Reflective::SemanticError);
   void set_base (in ::UmlCore::Classifier new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::FeatureSet available_feature ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_available_feature (in ::UmlCore::FeatureSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_available_feature ()
      raises (Reflective::SemanticError);
   UmlCollaborations::MessageSet message ()
```

```
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_message (in UmlCollaborations::MessageSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_message ()
      raises (Reflective::SemanticError);
   UmlCollaborations::MessageSet received_message ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_received_message (in UmlCollaborations::MessageSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_received_message ()
      raises (Reflective::SemanticError);
};

interface MessageClass : ::UmlCore::ModelElementClass {
   readonly attribute MessageUList all_of_kind_message;
   readonly attribute MessageUList all_of_type_message;
   Message create_message (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

 interface Message : MessageClass, ::UmlCore::ModelElement {
    UmlCollaborations::InteractionUList interaction ()
       raises (Reflective::NotSet, Reflective::SemanticError);
   void add_interaction (in UmlCollaborations::InteractionUList new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_interaction_before (in UmlCollaborations::Interaction
       new_value,
                                  in UmlCollaborations::Interaction before)
       raises (Reflective::StructuralError,
               Reflective::NotFound,
               Reflective::SemanticError);
    void remove_interaction ()
       raises (Reflective::SemanticError);
    Message activator ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void set_activator (in Message new_value)
       raises (Reflective::SemanticError);
    void unset_activator ()
       raises (Reflective::SemanticError);
    MessageUList activated_message ()
       raises (Reflective::NotSet, Reflective::SemanticError);
```

```
    void add_activated_message (in MessageUList new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_activated_message_before (in Message new_value,
                                   in Message before)
       raises (Reflective::StructuralError,
               Reflective::NotFound,
               Reflective::SemanticError);
    void remove_activated_message ()
       raises (Reflective::SemanticError);
    ::UmlCommonBehavior::Action action ()
       raises (Reflective::SemanticError);
    void set_action (in ::UmlCommonBehavior::Action new_value)
       raises (Reflective::SemanticError);
    ClassifierRole sender ()
       raises (Reflective::SemanticError);
    void set_sender (in ClassifierRole new_value)
       raises (Reflective::SemanticError);
    ClassifierRole receiver ()
       raises (Reflective::SemanticError);
    void set_receiver (in ClassifierRole new_value)
       raises (Reflective::SemanticError);
    MessageSet predecessor ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_predecessor (in MessageSet new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_predecessor ()
       raises (Reflective::SemanticError);
    MessageSet successor ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_successor (in MessageSet new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_successor ()
       raises (Reflective::SemanticError);
 };

 interface InteractionClass : ::UmlCore::ModelElementClass {
    readonly attribute InteractionUList all_of_kind_interaction;
    readonly attribute InteractionUList all_of_type_interaction;
    Interaction create_interaction (in ::UmlCore::Name name)
       raises (Reflective::SemanticError);
 };
```

```
interface Interaction : InteractionClass, ::UmlCore::ModelElement {
   UmlCollaborations::MessageSet message ()
      raises (Reflective::SemanticError);
   void add_message (in UmlCollaborations::MessageSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_message ()
      raises (Reflective::SemanticError);
   Collaboration uml_context ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_uml_context (in Collaboration new_value)
      raises (Reflective::SemanticError);
   void unset_uml_context ()
      raises (Reflective::SemanticError);
};

interface AssociationRoleClass : ::UmlCore::AssociationClass {
   readonly attribute AssociationRoleUList all_of_kind_association_role;
   readonly attribute AssociationRoleUList all_of_type_association_role;
   AssociationRole create_association_role (
                                    in ::UmlCore::Name name,
                                    in boolean is_root,
                                    in boolean is_leaf,
                                    in boolean is_abstract,
                               in ::UmlCore::Multiplicity multiplicity)
      raises (Reflective::SemanticError);
};

interface AssociationRole : AssociationRoleClass, ::UmlCore::Association {
   ::UmlCore::Multiplicity multiplicity ()
      raises (Reflective::SemanticError);
   void set_multiplicity (in ::UmlCore::Multiplicity new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::Association base ()
      raises (Reflective::SemanticError);
   void set_base (in ::UmlCore::Association new_value)
      raises (Reflective::SemanticError);
};

interface CollaborationClass : ::UmlCore::NamespaceClass {
   readonly attribute CollaborationUList all_of_kind_collaboration;
```

```
   readonly attribute CollaborationUList all_of_type_collaboration;
   Collaboration create_collaboration (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Collaboration : CollaborationClass, ::UmlCore::Namespace {
   UmlCollaborations::InteractionUList interaction ()
      raises (Reflective::NotSet, Reflective::SemanticError);
  void add_interaction (in UmlCollaborations::InteractionUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_interaction_before (in UmlCollaborations::Interaction
      new_value,
                                in UmlCollaborations::Interaction before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_interaction ()
      raises (Reflective::SemanticError);
   ::UmlCore::ModelElementSet constraining_element ()
      raises (Reflective::NotSet, Reflective::SemanticError);
  void add_constraining_element (in ::UmlCore::ModelElementSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_constraining_element ()
      raises (Reflective::SemanticError);
   ::UmlCore::Classifier represented_classifier ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_represented_classifier (in ::UmlCore::Classifier new_value)
      raises (Reflective::SemanticError);
   void unset_represented_classifier ()
      raises (Reflective::SemanticError);
   ::UmlCore::Operation represented_operation ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_represented_operation (in ::UmlCore::Operation new_value)
      raises (Reflective::SemanticError);
   void unset_represented_operation ()
      raises (Reflective::SemanticError);
};

struct InteractionContainsMessageLink {
   UmlCollaborations::Interaction interaction;
   UmlCollaborations::Message message;
```

```
};
typedef sequence <InteractionContainsMessageLink>
  InteractionContainsMessageLinkSet;

interface InteractionContainsMessage : Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   InteractionContainsMessageLinkSet
     all_interaction_contains_message_links();
   boolean exisits (in UmlCollaborations::Interaction interaction,
                    in UmlCollaborations::Message message);
   UmlCollaborations::InteractionUList
     with_message (in UmlCollaborations::Message message);
   UmlCollaborations::MessageSet with_interaction (
                          in UmlCollaborations::Interaction interaction);
   void add (in UmlCollaborations::Interaction interaction,
             in UmlCollaborations::Message message)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_interaction (in UmlCollaborations::Interaction
      interaction,
                                in UmlCollaborations::Message message,
                                in UmlCollaborations::Interaction before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
  void modify_interaction (in UmlCollaborations::Interaction interaction,
                          in UmlCollaborations::Message message,
                      in UmlCollaborations::Interaction new_interaction)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_message (in UmlCollaborations::Interaction interaction,
                        in UmlCollaborations::Message message,
                        in UmlCollaborations::Message new_message)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCollaborations::Interaction interaction,
                in UmlCollaborations::Message message)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
```

```
};

struct CollaborationOwnsInteractionLink {
   Collaboration uml_context;
   UmlCollaborations::Interaction interaction;
};
typedef sequence <CollaborationOwnsInteractionLink>
  CollaborationOwnsInteractionLinkSet;

interface CollaborationOwnsInteraction : Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   CollaborationOwnsInteractionLinkSet
     all_collaboration_owns_interaction_links();
   boolean exisits (in Collaboration uml_context,
                    in UmlCollaborations::Interaction interaction);
   Collaboration with_interaction (
                          in UmlCollaborations::Interaction interaction);
   UmlCollaborations::InteractionUList with_uml_context (
                                        in Collaboration uml_context);
   void add (in Collaboration uml_context,
             in UmlCollaborations::Interaction interaction)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_interaction (in Collaboration uml_context,
                          in UmlCollaborations::Interaction interaction,
                               in UmlCollaborations::Interaction before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_uml_context (in Collaboration uml_context,
                          in UmlCollaborations::Interaction interaction,
                           in Collaboration new_uml_context)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_interaction (in Collaboration uml_context,
                          in UmlCollaborations::Interaction interaction,
                      in UmlCollaborations::Interaction new_interaction)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Collaboration uml_context,
```

```
                  in UmlCollaborations::Interaction interaction)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct ClassifierRoleHasBaseOfClassifierLink {
   UmlCollaborations::ClassifierRole classifier_role;
   ::UmlCore::Classifier base;
};
typedef sequence <ClassifierRoleHasBaseOfClassifierLink>
  ClassifierRoleHasBaseOfClassifierLinkSet;

interface ClassifierRoleHasBaseOfClassifier : Reflective::RefAssociation {
    readonly attribute UmlCollaborationsPackage enclosing_package_ref;
    ClassifierRoleHasBaseOfClassifierLinkSet
      all_classifier_role_has_base_of_classifier_links();
    boolean exisits (in UmlCollaborations::ClassifierRole classifier_role,
                      in ::UmlCore::Classifier base);
    UmlCollaborations::ClassifierRoleSet with_base (
                                          in ::UmlCore::Classifier base);
    ::UmlCore::Classifier with_classifier_role (
                      in UmlCollaborations::ClassifierRole classifier_role);
    void add (in UmlCollaborations::ClassifierRole classifier_role,
               in ::UmlCore::Classifier base)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_classifier_role (
                      in UmlCollaborations::ClassifierRole classifier_role,
                      in ::UmlCore::Classifier base,
                  in UmlCollaborations::ClassifierRole new_classifier_role)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_base (in UmlCollaborations::ClassifierRole classifier_role,
                      in ::UmlCore::Classifier base,
                      in ::UmlCore::Classifier new_base)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in UmlCollaborations::ClassifierRole classifier_role,
                 in ::UmlCore::Classifier base)
```

```
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct AssociationEndRoleHasBaseOfAssociationEndLink {
   ::UmlCore::AssociationEnd base;
   UmlCollaborations::AssociationEndRole association_end_role;
};
typedef sequence <AssociationEndRoleHasBaseOfAssociationEndLink>
  AssociationEndRoleHasBaseOfAssociationEndLinkSet;

interface AssociationEndRoleHasBaseOfAssociationEnd :
                                            Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   AssociationEndRoleHasBaseOfAssociationEndLinkSet
     all_association_end_role_has_base_of_association_end_links();
   boolean
     exisits (in ::UmlCore::AssociationEnd base,
           in UmlCollaborations::AssociationEndRole association_end_role);
   ::UmlCore::AssociationEnd with_association_end_role (
           in UmlCollaborations::AssociationEndRole association_end_role);
   UmlCollaborations::AssociationEndRoleSet with_base (
                                      in ::UmlCore::AssociationEnd base);
   void add (in ::UmlCore::AssociationEnd base,
           in UmlCollaborations::AssociationEndRole association_end_role)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_base (
               in ::UmlCore::AssociationEnd base,
           in UmlCollaborations::AssociationEndRole association_end_role,
                     in ::UmlCore::AssociationEnd new_base)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_association_end_role (
           in ::UmlCore::AssociationEnd base,
           in UmlCollaborations::AssociationEndRole association_end_role,
           in UmlCollaborations::AssociationEndRole
      new_association_end_role)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
```

```
                  Reflective::NotFound);
       void remove (in ::UmlCore::AssociationEnd base,
                in UmlCollaborations::AssociationEndRole association_end_role)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
    };

    struct AssociationRoleHasBaseOfAssociationLink {
       ::UmlCore::Association base;
       UmlCollaborations::AssociationRole association_role;
    };
    typedef sequence <AssociationRoleHasBaseOfAssociationLink>
      AssociationRoleHasBaseOfAssociationLinkSet;

   interface AssociationRoleHasBaseOfAssociation : Reflective::RefAssociation
          {
       readonly attribute UmlCollaborationsPackage enclosing_package_ref;
       AssociationRoleHasBaseOfAssociationLinkSet
         all_association_role_has_base_of_association_links();
       boolean exisits (in ::UmlCore::Association base,
                      in UmlCollaborations::AssociationRole association_role);
       ::UmlCore::Association with_association_role (
                      in UmlCollaborations::AssociationRole association_role);
       UmlCollaborations::AssociationRoleSet with_base (
                                        in ::UmlCore::Association base);
       void add (in ::UmlCore::Association base,
                 in UmlCollaborations::AssociationRole association_role)
          raises (Reflective::StructuralError, Reflective::SemanticError);
       void modify_base (in ::UmlCore::Association base,
                       in UmlCollaborations::AssociationRole association_role,
                          in ::UmlCore::Association new_base)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
       void modify_association_role (
                        in ::UmlCore::Association base,
                       in UmlCollaborations::AssociationRole association_role,
                   in UmlCollaborations::AssociationRole new_association_role)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
```

```
              Reflective::NotFound);
   void remove (in ::UmlCore::Association base,
                in UmlCollaborations::AssociationRole association_role)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ClassifierRoleProvidesAvailableFeaturesLink {
   UmlCollaborations::ClassifierRole classifier_role;
   ::UmlCore::Feature available_feature;
};
typedef sequence <ClassifierRoleProvidesAvailableFeaturesLink>
ClassifierRoleProvidesAvailableFeaturesLinkSet;

interface ClassifierRoleProvidesAvailableFeatures :
                                            Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   ClassifierRoleProvidesAvailableFeaturesLinkSet
     all_classifier_role_provides_available_features_links();
   boolean exisits (in UmlCollaborations::ClassifierRole classifier_role,
                    in ::UmlCore::Feature available_feature);
   UmlCollaborations::ClassifierRoleSet with_available_feature (
                                  in ::UmlCore::Feature available_feature);
   ::UmlCore::FeatureSet with_classifier_role (
                    in UmlCollaborations::ClassifierRole classifier_role);
   void add (in UmlCollaborations::ClassifierRole classifier_role,
             in ::UmlCore::Feature available_feature)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_classifier_role (
                    in UmlCollaborations::ClassifierRole classifier_role,
                    in ::UmlCore::Feature available_feature,
                in UmlCollaborations::ClassifierRole new_classifier_role)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_available_feature (
                    in UmlCollaborations::ClassifierRole classifier_role,
                        in ::UmlCore::Feature available_feature,
                        in ::UmlCore::Feature new_available_feature)
      raises (Reflective::StructuralError,
```

```
                 Reflective::SemanticError,
                 Reflective::NotFound);
   void remove (in UmlCollaborations::ClassifierRole classifier_role,
                  in ::UmlCore::Feature available_feature)
      raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
};

struct CollaborationHasConstrainingModelElementLink {
   Collaboration constrained_collab;
   ::UmlCore::ModelElement constraining_element;
};
typedef sequence <CollaborationHasConstrainingModelElementLink>
CollaborationHasConstrainingModelElementLinkSet;

interface CollaborationHasConstrainingModelElement :
                                            Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   CollaborationHasConstrainingModelElementLinkSet
     all_collaboration_has_constraining_model_element_links();
   boolean exisits (in Collaboration constrained_collab,
                      in ::UmlCore::ModelElement constraining_element);
   CollaborationSet with_constraining_element (
                          in ::UmlCore::ModelElement constraining_element);
   ::UmlCore::ModelElementSet with_constrained_collab (
                                      in Collaboration constrained_collab);
   void add (in Collaboration constrained_collab,
             in ::UmlCore::ModelElement constraining_element)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_constrained_collab (
                              in Collaboration constrained_collab,
                           in ::UmlCore::ModelElement constraining_element,
                              in Collaboration new_constrained_collab)
      raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   void modify_constraining_element (
                           in Collaboration constrained_collab,
                           in ::UmlCore::ModelElement constraining_element,
                       in ::UmlCore::ModelElement new_constraining_element)
```

```
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
     void remove (in Collaboration constrained_collab,
                  in ::UmlCore::ModelElement constraining_element)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
  };

  struct MessageActivatesMessageLink {
     Message activator;
     Message activated_message;
  };
  typedef sequence <MessageActivatesMessageLink>
        MessageActivatesMessageLinkSet;

  interface MessageActivatesMessage : Reflective::RefAssociation {
     readonly attribute UmlCollaborationsPackage enclosing_package_ref;
     MessageActivatesMessageLinkSet all_message_activates_message_links();
     boolean exisits (in Message activator, in Message activated_message);
     Message with_activated_message (in Message activated_message);
     MessageUList with_activator (in Message activator);
     void add (in Message activator, in Message activated_message)
        raises (Reflective::StructuralError, Reflective::SemanticError);
     void add_before_activated_message (in Message activator,
                                        in Message activated_message,
                                        in Message before)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
     void modify_activator (in Message activator,
                            in Message activated_message,
                            in Message new_activator)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
     void modify_activated_message (in Message activator,
                                    in Message activated_message,
                                    in Message new_activated_message)
        raises (Reflective::StructuralError,
```

```
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in Message activator, in Message activated_message)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct CollaborationRepresentsClassifierLink {
      Collaboration representing_collaboration;
      ::UmlCore::Classifier represented_classifier;
   };
   typedef sequence <CollaborationRepresentsClassifierLink>
     CollaborationRepresentsClassifierLinkSet;

  interface CollaborationRepresentsClassifier : Reflective::RefAssociation {
      readonly attribute UmlCollaborationsPackage enclosing_package_ref;
      CollaborationRepresentsClassifierLinkSet
        all_collaboration_represents_classifier_links();
      boolean exisits (in Collaboration representing_collaboration,
                       in ::UmlCore::Classifier represented_classifier);
      CollaborationSet with_represented_classifier (
                           in ::UmlCore::Classifier represented_classifier);
      ::UmlCore::Classifier with_representing_collaboration (
                               in Collaboration representing_collaboration);
      void add (in Collaboration representing_collaboration,
                in ::UmlCore::Classifier represented_classifier)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_representing_collaboration (
                               in Collaboration representing_collaboration,
                            in ::UmlCore::Classifier represented_classifier,
                            in Collaboration new_representing_collaboration)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_represented_classifier (
                            in Collaboration representing_collaboration,
                            in ::UmlCore::Classifier represented_classifier,
                        in ::UmlCore::Classifier new_represented_classifier)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
```

```
                Reflective::NotFound);
   void remove (in Collaboration representing_collaboration,
                  in ::UmlCore::Classifier represented_classifier)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct CollaborationRepresentsOperationLink {
   Collaboration representing_collaboration;
   ::UmlCore::Operation represented_operation;
};
typedef sequence <CollaborationRepresentsOperationLink>
CollaborationRepresentsOperationLinkSet;

interface CollaborationRepresentsOperation : Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   CollaborationRepresentsOperationLinkSet
   all_collaboration_represents_operation_links();
   boolean exisits (in Collaboration representing_collaboration,
                    in ::UmlCore::Operation represented_operation);
   CollaborationSet with_represented_operation (
                          in ::UmlCore::Operation represented_operation);
   ::UmlCore::Operation with_representing_collaboration (
                            in Collaboration representing_collaboration);
   void add (in Collaboration representing_collaboration,
             in ::UmlCore::Operation represented_operation)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_representing_collaboration (
                            in Collaboration representing_collaboration,
                          in ::UmlCore::Operation represented_operation,
                        in Collaboration new_representing_collaboration)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_represented_operation (
                          in Collaboration representing_collaboration,
                          in ::UmlCore::Operation represented_operation,
                      in ::UmlCore::Operation new_represented_operation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
```

```
                 Reflective::NotFound);
      void remove (in Collaboration representing_collaboration,
                   in ::UmlCore::Operation represented_operation)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct MessageIsSentByActionLink {
      Message message0;
      ::UmlCommonBehavior::Action action;
   };
   typedef sequence <MessageIsSentByActionLink> MessageIsSentByActionLinkSet;

   interface MessageIsSentByAction : Reflective::RefAssociation {
      readonly attribute UmlCollaborationsPackage enclosing_package_ref;
      MessageIsSentByActionLinkSet all_message_is_sent_by_action_links();
      boolean exisits (in Message message0,
                       in ::UmlCommonBehavior::Action action);
      MessageSet with_action (in ::UmlCommonBehavior::Action action);
      ::UmlCommonBehavior::Action with_message0 (in Message message0);
      void add (in Message message0,
                in ::UmlCommonBehavior::Action action)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_message0 (in Message message0,
                            in ::UmlCommonBehavior::Action action,
                            in Message new_message0)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_action (in Message message0,
                          in ::UmlCommonBehavior::Action action,
                          in ::UmlCommonBehavior::Action new_action)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in Message message0,
                   in ::UmlCommonBehavior::Action action)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
```

```
};

struct MessageIsSentByClassifierRoleLink {
   UmlCollaborations::Message message;
   ClassifierRole sender;
};
typedef sequence <MessageIsSentByClassifierRoleLink>
  MessageIsSentByClassifierRoleLinkSet;

interface MessageIsSentByClassifierRole : Reflective::RefAssociation {
   readonly attribute UmlCollaborationsPackage enclosing_package_ref;
   MessageIsSentByClassifierRoleLinkSet
     all_message_is_sent_by_classifier_role_links();
   boolean exisits (in UmlCollaborations::Message message,
                    in ClassifierRole sender);
   UmlCollaborations::MessageSet with_sender (in ClassifierRole sender);
   ClassifierRole with_message (in UmlCollaborations::Message message);
   void add (in UmlCollaborations::Message message, in ClassifierRole
      sender)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_message (in UmlCollaborations::Message message,
                        in ClassifierRole sender,
                        in UmlCollaborations::Message new_message)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_sender (in UmlCollaborations::Message message,
                       in ClassifierRole sender,
                       in ClassifierRole new_sender)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCollaborations::Message message,
                in ClassifierRole sender)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct MessageIsReceivedByClassifierRoleLink {
   ClassifierRole receiver;
```

```
    Message received_message;
 };
 typedef sequence <MessageIsReceivedByClassifierRoleLink>
   MessageIsReceivedByClassifierRoleLinkSet;

interface MessageIsReceivedByClassifierRole : Reflective::RefAssociation {
    readonly attribute UmlCollaborationsPackage enclosing_package_ref;
    MessageIsReceivedByClassifierRoleLinkSet
      all_message_is_received_by_classifier_role_links();
    boolean exisits (in ClassifierRole receiver, in Message
        received_message);
    ClassifierRole with_received_message (in Message received_message);
    MessageSet with_receiver (in ClassifierRole receiver);
    void add (in ClassifierRole receiver, in Message received_message)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_receiver (in ClassifierRole receiver,
                          in Message received_message,
                          in ClassifierRole new_receiver)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_received_message (in ClassifierRole receiver,
                                  in Message received_message,
                                  in Message new_received_message)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in ClassifierRole receiver, in Message received_message)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
 };

 struct MessageHasPredecessorMessageLink {
    Message predecessor;
    Message successor;
 };
 typedef sequence <MessageHasPredecessorMessageLink>
   MessageHasPredecessorMessageLinkSet;

 interface MessageHasPredecessorMessage : Reflective::RefAssociation {
```

```
    readonly attribute UmlCollaborationsPackage enclosing_package_ref;
    MessageHasPredecessorMessageLinkSet
      all_message_has_predecessor_message_links();
    boolean exisits (in Message predecessor, in Message successor);
    MessageSet with_successor (in Message successor);
    MessageSet with_predecessor (in Message predecessor);
    void add (in Message predecessor, in Message successor)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_predecessor (in Message predecessor,
                             in Message successor,
                             in Message new_predecessor)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_successor (in Message predecessor,
                           in Message successor,
                           in Message new_successor)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in Message predecessor, in Message successor)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
};

interface UmlCollaborationsPackageFactory {
    UmlCollaborationsPackage create_uml_collaborations_package ()
       raises (Reflective::SemanticError);
};

interface UmlCollaborationsPackage : Reflective::RefPackage {
    readonly attribute AssociationEndRoleClass
       association_end_role_class_ref;
    readonly attribute ClassifierRoleClass classifier_role_class_ref;
    readonly attribute MessageClass message_class_ref;
    readonly attribute InteractionClass interaction_class_ref;
    readonly attribute AssociationRoleClass association_role_class_ref;
    readonly attribute CollaborationClass collaboration_class_ref;

    readonly attribute InteractionContainsMessage
```

```
        interaction_contains_message_ref;
      readonly attribute CollaborationOwnsInteraction
        collaboration_owns_interaction_ref;
      readonly attribute ClassifierRoleHasBaseOfClassifier
        classifier_role_has_base_of_classifier_ref;
      readonly attribute AssociationEndRoleHasBaseOfAssociationEnd
        association_end_role_has_base_of_association_end_ref;
      readonly attribute AssociationRoleHasBaseOfAssociation
        association_role_has_base_of_association_ref;
      readonly attribute ClassifierRoleProvidesAvailableFeatures
        classifier_role_provides_available_features_ref;
      readonly attribute CollaborationHasConstrainingModelElement
        collaboration_has_constraining_model_element_ref;
      readonly attribute MessageActivatesMessage
         message_activates_message_ref;
      readonly attribute CollaborationRepresentsClassifier
        collaboration_represents_classifier_ref;
      readonly attribute CollaborationRepresentsOperation
        collaboration_represents_operation_ref;
     readonly attribute MessageIsSentByAction message_is_sent_by_action_ref;
      readonly attribute MessageIsSentByClassifierRole
        message_is_sent_by_classifier_role_ref;
      readonly attribute MessageIsReceivedByClassifierRole
        message_is_received_by_classifier_role_ref;
      readonly attribute MessageHasPredecessorMessage
        message_has_predecessor_message_ref;
  };
};
```

5.4.5 UMLCommonBehavior

```
#include "UmlCore.idl"

module UmlCommonBehavior {
    interface UmlCommonBehaviorPackage;
    interface LinkEnd;
    interface LinkEndClass;
    typedef sequence<LinkEnd> LinkEndUList;
    typedef sequence<LinkEnd> LinkEndSet;
    interface Action;
    interface ActionClass;
    typedef sequence<Action> ActionUList;
    typedef sequence<Action> ActionSet;
    interface UmlObject;
    interface UmlObjectClass;
    typedef sequence<UmlObject> UmlObjectUList;
    interface DataValue;
    interface DataValueClass;
    typedef sequence<DataValue> DataValueUList;
    interface UmlInstance;
    interface UmlInstanceClass;
    typedef sequence<UmlInstance> UmlInstanceUList;
    typedef sequence<UmlInstance> UmlInstanceSet;
    interface ReturnAction;
    interface ReturnActionClass;
    typedef sequence<ReturnAction> ReturnActionUList;
    interface ActionSequence;
    interface ActionSequenceClass;
    typedef sequence<ActionSequence> ActionSequenceUList;
    interface LocalInvocation;
    interface LocalInvocationClass;
    typedef sequence<LocalInvocation> LocalInvocationUList;
    interface Reception;
    interface ReceptionClass;
    typedef sequence<Reception> ReceptionUList;
    typedef sequence<Reception> ReceptionSet;
    interface Signal;
    interface SignalClass;
    typedef sequence<Signal> SignalUList;
    interface TerminateAction;
```

```
interface TerminateActionClass;
typedef sequence<TerminateAction> TerminateActionUList;
interface MessageInstance;
interface MessageInstanceClass;
typedef sequence<MessageInstance> MessageInstanceUList;
typedef sequence<MessageInstance> MessageInstanceSet;
interface Argument;
interface ArgumentClass;
typedef sequence<Argument> ArgumentUList;
interface CallAction;
interface CallActionClass;
typedef sequence<CallAction> CallActionUList;
interface CreateAction;
interface CreateActionClass;
typedef sequence<CreateAction> CreateActionUList;
typedef sequence<CreateAction> CreateActionSet;
interface UninterpretedAction;
interface UninterpretedActionClass;
typedef sequence<UninterpretedAction> UninterpretedActionUList;
interface Call;
interface CallClass;
typedef sequence<Call> CallUList;
interface Link;
interface LinkClass;
typedef sequence<Link> LinkUList;
typedef sequence<Link> LinkSet;
interface SendAction;
interface SendActionClass;
typedef sequence<SendAction> SendActionUList;
interface AttributeLink;
interface AttributeLinkClass;
typedef sequence<AttributeLink> AttributeLinkUList;
typedef sequence<AttributeLink> AttributeLinkSet;
interface UmlException;
interface UmlExceptionClass;
typedef sequence<UmlException> UmlExceptionUList;
typedef sequence<UmlException> UmlExceptionSet;
interface DestroyAction;
interface DestroyActionClass;
typedef sequence<DestroyAction> DestroyActionUList;
interface LinkObject;
```

```
interface LinkObjectClass;
typedef sequence<LinkObject> LinkObjectUList;

interface CallClass : ::UmlCore::ModelElementClass {
   readonly attribute CallUList all_of_kind_call;
   readonly attribute CallUList all_of_type_call;
   Call create_call (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Call : CallClass, ::UmlCore::ModelElement { };

interface LinkEndClass : ::UmlCore::ModelElementClass {
   readonly attribute LinkEndUList all_of_kind_link_end;
   readonly attribute LinkEndUList all_of_type_link_end;
   LinkEnd create_link_end (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface LinkEnd : LinkEndClass, ::UmlCore::ModelElement {
   UmlCommonBehavior::UmlInstance uml_instance ()
      raises (Reflective::SemanticError);
   void set_uml_instance (in UmlCommonBehavior::UmlInstance new_value)
      raises (Reflective::SemanticError);
   Link owning_link ()
      raises (Reflective::SemanticError);
   void set_owning_link (in Link new_value)
      raises (Reflective::SemanticError);
   void add_owning_link_before (in Link new_value, in Link before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   ::UmlCore::AssociationEnd association_end ()
      raises (Reflective::SemanticError);
   void set_association_end (in ::UmlCore::AssociationEnd new_value)
      raises (Reflective::SemanticError);
};

interface LinkClass : ::UmlCore::ModelElementClass {
   readonly attribute LinkUList all_of_kind_link;
   readonly attribute LinkUList all_of_type_link;
```

```
   Link create_link (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Link : LinkClass, ::UmlCore::ModelElement {
   ::UmlCore::Association association ()
      raises (Reflective::SemanticError);
   void set_association (in ::UmlCore::Association new_value)
      raises (Reflective::SemanticError);
   LinkEndSet link_role ()
      raises (Reflective::SemanticError);
   void add_link_role (in LinkEndSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_link_role (in LinkEnd old_value, in LinkEnd new_value)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_link_role ()
      raises (Reflective::StructuralError, Reflective::SemanticError);
};

interface AttributeLinkClass : ::UmlCore::ModelElementClass {
   readonly attribute AttributeLinkUList all_of_kind_attribute_link;
   readonly attribute AttributeLinkUList all_of_type_attribute_link;
   AttributeLink create_attribute_link (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface AttributeLink : AttributeLinkClass, ::UmlCore::ModelElement {
   ::UmlCore::UmlAttribute uml_attribute ()
      raises (Reflective::SemanticError);
   void set_uml_attribute (in ::UmlCore::UmlAttribute new_value)
      raises (Reflective::SemanticError);
   UmlInstance uml_value ()
      raises (Reflective::SemanticError);
   void set_uml_value (in UmlInstance new_value)
      raises (Reflective::SemanticError);
   UmlInstance owning_instance ()
      raises (Reflective::SemanticError);
   void set_owning_instance (in UmlInstance new_value)
      raises (Reflective::SemanticError);
```

```
      void add_owning_instance_before (in UmlInstance new_value,
                                        in UmlInstance before)
         raises (Reflective::StructuralError,
                 Reflective::NotFound,
                 Reflective::SemanticError);
   };

   interface UmlInstanceClass : ::UmlCore::ModelElementClass {
      readonly attribute UmlInstanceUList all_of_kind_uml_instance;
      readonly attribute UmlInstanceUList all_of_type_uml_instance;
      UmlInstance create_uml_instance (in ::UmlCore::Name name)
         raises (Reflective::SemanticError);
   };

   interface UmlInstance : UmlInstanceClass, ::UmlCore::ModelElement {
      ::UmlCore::ClassifierSet classifier ()
         raises (Reflective::SemanticError);
      void add_classifier (in ::UmlCore::ClassifierSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_classifier ()
         raises (Reflective::SemanticError);
      AttributeLinkSet slot ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_slot (in AttributeLinkSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_slot ()
         raises (Reflective::SemanticError);
      UmlCommonBehavior::LinkEndSet link_end ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_link_end (in UmlCommonBehavior::LinkEndSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_link_end ()
         raises (Reflective::SemanticError);
      MessageInstanceSet received_message_instance ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_received_message_instance (in MessageInstanceSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_received_message_instance ()
         raises (Reflective::SemanticError);
      AttributeLinkSet owned_attribute_link ()
         raises (Reflective::NotSet, Reflective::SemanticError);
```

```
   void add_owned_attribute_link (in AttributeLinkSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_owned_attribute_link ()
      raises (Reflective::SemanticError);
   MessageInstanceSet argument_owner ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_argument_owner (in MessageInstanceSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_argument_owner ()
      raises (Reflective::SemanticError);
   MessageInstanceSet sent_message_instance ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_sent_message_instance (in MessageInstanceSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_sent_message_instance ()
      raises (Reflective::SemanticError);
};

interface UmlObjectClass : UmlInstanceClass {
   readonly attribute UmlObjectUList all_of_kind_uml_object;
   readonly attribute UmlObjectUList all_of_type_uml_object;
   UmlObject create_uml_object (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface UmlObject : UmlObjectClass, UmlInstance { };

interface MessageInstanceClass : ::UmlCore::ModelElementClass {
   readonly attribute MessageInstanceUList all_of_kind_message_instance;
   readonly attribute MessageInstanceUList all_of_type_message_instance;
   MessageInstance create_message_instance (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface MessageInstance : MessageInstanceClass, ::UmlCore::ModelElement {
   ::UmlCore::Request specification ()
      raises (Reflective::SemanticError);
   void set_specification (in ::UmlCore::Request new_value)
      raises (Reflective::SemanticError);
   UmlInstance receiver ()
      raises (Reflective::SemanticError);
```

```
    void set_receiver (in UmlInstance new_value)
      raises (Reflective::SemanticError);
    UmlInstanceSet argument ()
      raises (Reflective::NotSet, Reflective::SemanticError);
    void add_argument (in UmlInstanceSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_argument ()
      raises (Reflective::SemanticError);
    UmlInstance sender ()
      raises (Reflective::SemanticError);
    void set_sender (in UmlInstance new_value)
      raises (Reflective::SemanticError);
  };

  interface DataValueClass : UmlInstanceClass {
    readonly attribute DataValueUList all_of_kind_data_value;
    readonly attribute DataValueUList all_of_type_data_value;
    DataValue create_data_value (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
  };

  interface DataValue : DataValueClass, UmlInstance { };

  interface SignalClass : ::UmlCore::GeneralizableElementClass,
                          ::UmlCore::RequestClass {
    readonly attribute SignalUList all_of_kind_signal;
    readonly attribute SignalUList all_of_type_signal;
    Signal create_signal (in ::UmlCore::Name name,
                          in boolean is_root,
                          in boolean is_leaf,
                          in boolean is_abstract)
      raises (Reflective::SemanticError);
  };

  interface Signal : SignalClass, ::UmlCore::GeneralizableElement,
                                  ::UmlCore::Request {
    UmlCommonBehavior::ReceptionSet reception ()
      raises (Reflective::NotSet, Reflective::SemanticError);
    void add_reception (in UmlCommonBehavior::ReceptionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_reception ()
```

```
      raises (Reflective::SemanticError);
   ::UmlCore::ParameterUList parameter ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_parameter (in ::UmlCore::ParameterUList new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_parameter_before (in ::UmlCore::Parameter new_value,
                              in ::UmlCore::Parameter before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void remove_parameter ()
      raises (Reflective::SemanticError);
};

interface UmlExceptionClass : SignalClass {
   readonly attribute UmlExceptionUList all_of_kind_uml_exception;
   readonly attribute UmlExceptionUList all_of_type_uml_exception;
   UmlException create_uml_exception (in ::UmlCore::Name name,
                                      in boolean is_root,
                                      in boolean is_leaf,
                                      in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface UmlException : UmlExceptionClass, Signal {
   ::UmlCore::BehavioralFeatureSet uml_context ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_uml_context (in ::UmlCore::BehavioralFeatureSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_uml_context ()
      raises (Reflective::SemanticError);
};

interface ReceptionClass : ::UmlCore::BehavioralFeatureClass {
   readonly attribute ReceptionUList all_of_kind_reception;
   readonly attribute ReceptionUList all_of_type_reception;
   Reception create_reception (in ::UmlCore::Name name,
                               in ::UmlCore::ScopeKind owner_scope,
                               in ::UmlCore::VisibilityKind visibility,
                               in boolean is_query,
                               in boolean is_polymorphic,
```

```
                                in ::UmlCore::Uninterpreted specification)
      raises (Reflective::SemanticError);
};

interface Reception : ReceptionClass, ::UmlCore::BehavioralFeature {
   boolean is_polymorphic ()
      raises (Reflective::SemanticError);
   void set_is_polymorphic (in boolean new_value)
      raises (Reflective::SemanticError);
   ::UmlCore::Uninterpreted specification ()
      raises (Reflective::SemanticError);
   void set_specification (in ::UmlCore::Uninterpreted new_value)
      raises (Reflective::SemanticError);
   UmlCommonBehavior::Signal signal ()
      raises (Reflective::SemanticError);
   void set_signal (in UmlCommonBehavior::Signal new_value)
      raises (Reflective::SemanticError);
};

interface ArgumentClass : ::UmlCore::ModelElementClass {
   readonly attribute ArgumentUList all_of_kind_argument;
   readonly attribute ArgumentUList all_of_type_argument;
   Argument create_argument (in ::UmlCore::Name name,
                             in ::UmlCore::Expression uml_value)
      raises (Reflective::SemanticError);
};

interface Argument : ArgumentClass, ::UmlCore::ModelElement {
   ::UmlCore::Expression uml_value ()
      raises (Reflective::SemanticError);
   void set_uml_value (in ::UmlCore::Expression new_value)
      raises (Reflective::SemanticError);
   Action owning_action ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_owning_action (in Action new_value)
      raises (Reflective::SemanticError);
   void unset_owning_action ()
      raises (Reflective::SemanticError);
};

interface ActionClass : ::UmlCore::ModelElementClass {
```

```
    readonly attribute ActionUList all_of_kind_action;
    readonly attribute ActionUList all_of_type_action;
    Action create_action (in ::UmlCore::Name name,
                          in ::UmlCore::Expression recurrence,
                          in ::UmlCore::ObjectSetExpression target,
                          in boolean is_asynchronous,
                          in string script)
       raises (Reflective::SemanticError);
 };

 interface Action : ActionClass, ::UmlCore::ModelElement {
    ::UmlCore::Expression recurrence ()
       raises (Reflective::SemanticError);
    void set_recurrence (in ::UmlCore::Expression new_value)
       raises (Reflective::SemanticError);
    ::UmlCore::ObjectSetExpression target ()
       raises (Reflective::SemanticError);
    void set_target (in ::UmlCore::ObjectSetExpression new_value)
       raises (Reflective::SemanticError);
    boolean is_asynchronous ()
       raises (Reflective::SemanticError);
    void set_is_asynchronous (in boolean new_value)
       raises (Reflective::SemanticError);
    string script ()
       raises (Reflective::SemanticError);
    void set_script (in string new_value)
       raises (Reflective::SemanticError);
    ArgumentUList actual_argument ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void add_actual_argument (in ArgumentUList new_value)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_actual_argument_before (in Argument new_value,
                                     in Argument before)
       raises (Reflective::StructuralError,
               Reflective::NotFound,
               Reflective::SemanticError);
    void remove_actual_argument ()
       raises (Reflective::SemanticError);
    ::UmlCore::Request message ()
       raises (Reflective::NotSet, Reflective::SemanticError);
    void set_message (in ::UmlCore::Request new_value)
```

```
      raises (Reflective::SemanticError);
   void unset_message ()
      raises (Reflective::SemanticError);
   UmlCommonBehavior::ActionSequence action_sequence ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_action_sequence (in UmlCommonBehavior::ActionSequence
      new_value)
      raises (Reflective::SemanticError);
   void unset_action_sequence ()
      raises (Reflective::SemanticError);
};

interface CallActionClass : ActionClass {
   readonly attribute CallActionUList all_of_kind_call_action;
   readonly attribute CallActionUList all_of_type_call_action;
   CallAction create_call_action (in ::UmlCore::Name name,
                                  in ::UmlCore::Expression recurrence,
                               in ::UmlCore::ObjectSetExpression target,
                                  in boolean is_asynchronous,
                                  in string script,
                                  in ::UmlCore::SynchronousKind mode)
      raises (Reflective::SemanticError);
};

interface CallAction : CallActionClass, Action {
   ::UmlCore::SynchronousKind mode ()
      raises (Reflective::SemanticError);
   void set_mode (in ::UmlCore::SynchronousKind new_value)
      raises (Reflective::SemanticError);
};

interface CreateActionClass : ActionClass {
   readonly attribute CreateActionUList all_of_kind_create_action;
   readonly attribute CreateActionUList all_of_type_create_action;
   CreateAction create_create_action (
                                    in ::UmlCore::Name name,
                                    in ::UmlCore::Expression recurrence,
                               in ::UmlCore::ObjectSetExpression target,
                                    in boolean is_asynchronous,
                                    in string script)
      raises (Reflective::SemanticError);
```

```
};

interface CreateAction : CreateActionClass, Action {
   ::UmlCore::Classifier instantiation ()
      raises (Reflective::SemanticError);
   void set_instantiation (in ::UmlCore::Classifier new_value)
      raises (Reflective::SemanticError);
};

interface DestroyActionClass : ActionClass {
   readonly attribute DestroyActionUList all_of_kind_destroy_action;
   readonly attribute DestroyActionUList all_of_type_destroy_action;
   DestroyAction create_destroy_action (
                                 in ::UmlCore::Name name,
                                 in ::UmlCore::Expression recurrence,
                             in ::UmlCore::ObjectSetExpression target,
                                 in boolean is_asynchronous,
                                 in string script)
      raises (Reflective::SemanticError);
};

interface DestroyAction : DestroyActionClass, Action { };

interface LocalInvocationClass : ActionClass {
   readonly attribute LocalInvocationUList all_of_kind_local_invocation;
   readonly attribute LocalInvocationUList all_of_type_local_invocation;
   LocalInvocation create_local_invocation (
                                 in ::UmlCore::Name name,
                                 in ::UmlCore::Expression recurrence,
                             in ::UmlCore::ObjectSetExpression target,
                                 in boolean is_asynchronous,
                                 in string script)
      raises (Reflective::SemanticError);
};

interface LocalInvocation : LocalInvocationClass, Action { };

interface SendActionClass : ActionClass {
   readonly attribute SendActionUList all_of_kind_send_action;
   readonly attribute SendActionUList all_of_type_send_action;
   SendAction create_send_action (in ::UmlCore::Name name,
```

```
                                    in ::UmlCore::Expression recurrence,
                                 in ::UmlCore::ObjectSetExpression target,
                                    in boolean is_asynchronous,
                                    in string script)
      raises (Reflective::SemanticError);
};

interface SendAction : SendActionClass, Action { };

interface ReturnActionClass : ActionClass {
   readonly attribute ReturnActionUList all_of_kind_return_action;
   readonly attribute ReturnActionUList all_of_type_return_action;
   ReturnAction create_return_action (
                                    in ::UmlCore::Name name,
                                    in ::UmlCore::Expression recurrence,
                                 in ::UmlCore::ObjectSetExpression target,
                                    in boolean is_asynchronous,
                                    in string script)
      raises (Reflective::SemanticError);
};

interface ReturnAction : ReturnActionClass, Action { };

interface TerminateActionClass : ActionClass {
   readonly attribute TerminateActionUList all_of_kind_terminate_action;
   readonly attribute TerminateActionUList all_of_type_terminate_action;
   TerminateAction create_terminate_action (
                                    in ::UmlCore::Name name,
                                    in ::UmlCore::Expression recurrence,
                                 in ::UmlCore::ObjectSetExpression target,
                                    in boolean is_asynchronous,
                                    in string script)
      raises (Reflective::SemanticError);
};

interface TerminateAction : TerminateActionClass, Action { };

interface UninterpretedActionClass : ActionClass {
   readonly attribute UninterpretedActionUList
     all_of_kind_uninterpreted_action;
   readonly attribute UninterpretedActionUList
```

```
      all_of_type_uninterpreted_action;
    UninterpretedAction create_uninterpreted_action (
                                        in ::UmlCore::Name name,
                                        in ::UmlCore::Expression recurrence,
                                    in ::UmlCore::ObjectSetExpression target,
                                        in boolean is_asynchronous,
                                        in string script)
      raises (Reflective::SemanticError);
};

interface UninterpretedAction : UninterpretedActionClass, Action { };

interface ActionSequenceClass : ::UmlCore::ModelElementClass {
    readonly attribute ActionSequenceUList all_of_kind_action_sequence;
    readonly attribute ActionSequenceUList all_of_type_action_sequence;
    ActionSequence create_action_sequence (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface ActionSequence : ActionSequenceClass, ::UmlCore::ModelElement {
    UmlCommonBehavior::ActionSet action ()
      raises (Reflective::NotSet, Reflective::SemanticError);
    void add_action (in UmlCommonBehavior::ActionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_action ()
      raises (Reflective::SemanticError);
};

interface LinkObjectClass : UmlObjectClass, LinkClass {
    readonly attribute LinkObjectUList all_of_kind_link_object;
    readonly attribute LinkObjectUList all_of_type_link_object;
    LinkObject create_link_object (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface LinkObject : LinkObjectClass, UmlObject, Link { };

struct InstanceInstantiatesClassifierLink {
    UmlInstance instantiated_instance;
    ::UmlCore::Classifier classifier;
};
```

```
typedef sequence <InstanceInstantiatesClassifierLink>
  InstanceInstantiatesClassifierLinkSet;

interface InstanceInstantiatesClassifier : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   InstanceInstantiatesClassifierLinkSet
     all_instance_instantiates_classifier_links();
   boolean exists (in UmlInstance instantiated_instance,
                   in ::UmlCore::Classifier classifier);
   UmlInstanceSet with_classifier (in ::UmlCore::Classifier classifier);
   ::UmlCore::ClassifierSet with_instantiated_instance (
                                   in UmlInstance instantiated_instance);
   void add (in UmlInstance instantiated_instance,
             in ::UmlCore::Classifier classifier)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_instantiated_instance (
                                   in UmlInstance instantiated_instance,
                                   in ::UmlCore::Classifier classifier,
                                in UmlInstance new_instantiated_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_classifier (in UmlInstance instantiated_instance,
                           in ::UmlCore::Classifier classifier,
                           in ::UmlCore::Classifier new_classifier)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlInstance instantiated_instance,
                in ::UmlCore::Classifier classifier)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ActionOwnsArgumentLink {
   Argument actual_argument;
   Action owning_action;
};
typedef sequence <ActionOwnsArgumentLink> ActionOwnsArgumentLinkSet;
```

```
interface ActionOwnsArgument : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   ActionOwnsArgumentLinkSet all_action_owns_argument_links();
   boolean exists (in Argument actual_argument, in Action owning_action);
   ArgumentUList with_owning_action (in Action owning_action);
   Action with_actual_argument (in Argument actual_argument);
   void add (in Argument actual_argument, in Action owning_action)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void add_before_actual_argument (in Argument actual_argument,
                                    in Action owning_action,
                                    in Argument before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_actual_argument (in Argument actual_argument,
                                in Action owning_action,
                                in Argument new_actual_argument)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_owning_action (in Argument actual_argument,
                              in Action owning_action,
                              in Action new_owning_action)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Argument actual_argument, in Action owning_action)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct CreateActionInstantiatesClassifierLink {
   UmlCommonBehavior::CreateAction create_action;
   ::UmlCore::Classifier instantiation;
};
typedef sequence <CreateActionInstantiatesClassifierLink>
  CreateActionInstantiatesClassifierLinkSet;

interface CreateActionInstantiatesClassifier : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
```

```
    CreateActionInstantiatesClassifierLinkSet
      all_create_action_instantiates_classifier_links();
    boolean exists (in UmlCommonBehavior::CreateAction create_action,
                    in ::UmlCore::Classifier instantiation);
    UmlCommonBehavior::CreateActionSet with_instantiation (
                                  in ::UmlCore::Classifier instantiation);
    ::UmlCore::Classifier with_create_action (
                           in UmlCommonBehavior::CreateAction create_action);
    void add (in UmlCommonBehavior::CreateAction create_action,
              in ::UmlCore::Classifier instantiation)
      raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_create_action (
                           in UmlCommonBehavior::CreateAction create_action,
                           in ::UmlCore::Classifier instantiation,
                        in UmlCommonBehavior::CreateAction new_create_action)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
    void modify_instantiation (
                           in UmlCommonBehavior::CreateAction create_action,
                               in ::UmlCore::Classifier instantiation,
                               in ::UmlCore::Classifier new_instantiation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
    void remove (in UmlCommonBehavior::CreateAction create_action,
                 in ::UmlCore::Classifier instantiation)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
  };

  struct AttributeLinkIsInstanceOfAttributeLink {
    AttributeLink instance;
    ::UmlCore::UmlAttribute uml_attribute;
  };
  typedef sequence <AttributeLinkIsInstanceOfAttributeLink>
    AttributeLinkIsInstanceOfAttributeLinkSet;

 interface AttributeLinkIsInstanceOfAttribute : Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
```

```
    AttributeLinkIsInstanceOfAttributeLinkSet
      all_attribute_link_is_instance_of_attribute_links();
    boolean exists (in AttributeLink instance,
                    in ::UmlCore::UmlAttribute uml_attribute);
    AttributeLinkSet with_uml_attribute (in ::UmlCore::UmlAttribute
                                         uml_attribute);
    ::UmlCore::UmlAttribute with_instance (in AttributeLink instance);
    void add (in AttributeLink instance,
              in ::UmlCore::UmlAttribute uml_attribute)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_instance (in AttributeLink instance,
                          in ::UmlCore::UmlAttribute uml_attribute,
                          in AttributeLink new_instance)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_uml_attribute (in AttributeLink instance,
                               in ::UmlCore::UmlAttribute uml_attribute,
                             in ::UmlCore::UmlAttribute new_uml_attribute)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in AttributeLink instance,
                 in ::UmlCore::UmlAttribute uml_attribute)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
};

struct AttributeLilnkHasValueOfLinkLink {
    AttributeLink slot;
    UmlInstance uml_value;
};
typedef sequence <AttributeLilnkHasValueOfLinkLink>
  AttributeLilnkHasValueOfLinkLinkSet;

interface AttributeLilnkHasValueOfLink : Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
    AttributeLilnkHasValueOfLinkLinkSet
      all_attribute_lilnk_has_value_of_link_links();
    boolean exists (in AttributeLink slot, in UmlInstance uml_value);
```

```
   AttributeLinkSet with_uml_value (in UmlInstance uml_value);
   UmlInstance with_slot (in AttributeLink slot);
   void add (in AttributeLink slot, in UmlInstance uml_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_slot (in AttributeLink slot,
                     in UmlInstance uml_value,
                     in AttributeLink new_slot)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_uml_value (in AttributeLink slot,
                          in UmlInstance uml_value,
                          in UmlInstance new_uml_value)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in AttributeLink slot, in UmlInstance uml_value)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct LinkEndIsOfTypeInstanceLink {
   UmlCommonBehavior::UmlInstance uml_instance;
   UmlCommonBehavior::LinkEnd link_end;
};
typedef sequence <LinkEndIsOfTypeInstanceLink>
      LinkEndIsOfTypeInstanceLinkSet;

interface LinkEndIsOfTypeInstance : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
  LinkEndIsOfTypeInstanceLinkSet all_link_end_is_of_type_instance_links();
   boolean exists (in UmlCommonBehavior::UmlInstance uml_instance,
                   in UmlCommonBehavior::LinkEnd link_end);
   UmlCommonBehavior::UmlInstance with_link_end (
                                  in UmlCommonBehavior::LinkEnd link_end);
   UmlCommonBehavior::LinkEndSet with_uml_instance (
                          in UmlCommonBehavior::UmlInstance uml_instance);
   void add (in UmlCommonBehavior::UmlInstance uml_instance,
             in UmlCommonBehavior::LinkEnd link_end)
      raises (Reflective::StructuralError, Reflective::SemanticError);
```

```
   void modify_uml_instance (
                          in UmlCommonBehavior::UmlInstance uml_instance,
                          in UmlCommonBehavior::LinkEnd link_end,
                      in UmlCommonBehavior::UmlInstance new_uml_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_link_end (in UmlCommonBehavior::UmlInstance uml_instance,
                         in UmlCommonBehavior::LinkEnd link_end,
                         in UmlCommonBehavior::LinkEnd new_link_end)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlCommonBehavior::UmlInstance uml_instance,
                in UmlCommonBehavior::LinkEnd link_end)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ReceptionFeatureReceivesSignalLink {
   UmlCommonBehavior::Signal signal;
   UmlCommonBehavior::Reception reception;
};
typedef sequence <ReceptionFeatureReceivesSignalLink>
  ReceptionFeatureReceivesSignalLinkSet;

interface ReceptionFeatureReceivesSignal : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   ReceptionFeatureReceivesSignalLinkSet
     all_reception_feature_receives_signal_links();
   boolean exists (in UmlCommonBehavior::Signal signal,
                   in UmlCommonBehavior::Reception reception);
   UmlCommonBehavior::Signal with_reception (
                              in UmlCommonBehavior::Reception reception);
   UmlCommonBehavior::ReceptionSet with_signal (
                                    in UmlCommonBehavior::Signal signal);
   void add (in UmlCommonBehavior::Signal signal,
             in UmlCommonBehavior::Reception reception)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_signal (in UmlCommonBehavior::Signal signal,
```

```
                            in UmlCommonBehavior::Reception reception,
                            in UmlCommonBehavior::Signal new_signal)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_reception (in UmlCommonBehavior::Signal signal,
                             in UmlCommonBehavior::Reception reception,
                             in UmlCommonBehavior::Reception new_reception)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in UmlCommonBehavior::Signal signal,
                   in UmlCommonBehavior::Reception reception)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct SignalOwnsParameterLink {
      UmlCommonBehavior::Signal signal;
      ::UmlCore::Parameter parameter;
   };
   typedef sequence <SignalOwnsParameterLink> SignalOwnsParameterLinkSet;

   interface SignalOwnsParameter : Reflective::RefAssociation {
      readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
      SignalOwnsParameterLinkSet all_signal_owns_parameter_links();
      boolean exists (in UmlCommonBehavior::Signal signal,
                      in ::UmlCore::Parameter parameter);
      UmlCommonBehavior::Signal with_parameter (
                                         in ::UmlCore::Parameter parameter);
      ::UmlCore::ParameterUList with_signal (
                                       in UmlCommonBehavior::Signal signal);
      void add (in UmlCommonBehavior::Signal signal,
                in ::UmlCore::Parameter parameter)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void add_before_parameter (in UmlCommonBehavior::Signal signal,
                                 in ::UmlCore::Parameter parameter,
                                 in ::UmlCore::Parameter before)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
```

```
                  Reflective::NotFound);
      void modify_signal (in UmlCommonBehavior::Signal signal,
                          in ::UmlCore::Parameter parameter,
                          in UmlCommonBehavior::Signal new_signal)
         raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
      void modify_parameter (in UmlCommonBehavior::Signal signal,
                             in ::UmlCore::Parameter parameter,
                             in ::UmlCore::Parameter new_parameter)
         raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
      void remove (in UmlCommonBehavior::Signal signal,
                   in ::UmlCore::Parameter parameter)
         raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
   };

   struct ActionIsInitiatedByRequestLink {
      ::UmlCore::Request message;
      UmlCommonBehavior::Action action;
   };
   typedef sequence <ActionIsInitiatedByRequestLink>
     ActionIsInitiatedByRequestLinkSet;

   interface ActionIsInitiatedByRequest : Reflective::RefAssociation {
      readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
      ActionIsInitiatedByRequestLinkSet
        all_action_is_initiated_by_request_links();
      boolean exists (in ::UmlCore::Request message,
                      in UmlCommonBehavior::Action action);
      ::UmlCore::Request with_action (in UmlCommonBehavior::Action action);
      UmlCommonBehavior::ActionSet with_message (in ::UmlCore::Request
         message);
      void add (in ::UmlCore::Request message,
                in UmlCommonBehavior::Action action)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_message (in ::UmlCore::Request message,
                           in UmlCommonBehavior::Action action,
```

```
                        in ::UmlCore::Request new_message)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_action (in ::UmlCore::Request message,
                       in UmlCommonBehavior::Action action,
                       in UmlCommonBehavior::Action new_action)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCore::Request message,
                in UmlCommonBehavior::Action action)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct MessageInstanceIsSpecifiedByRequestLink {
   MessageInstance instance;
   ::UmlCore::Request specification;
};
typedef sequence <MessageInstanceIsSpecifiedByRequestLink>
  MessageInstanceIsSpecifiedByRequestLinkSet;

interface MessageInstanceIsSpecifiedByRequest : Reflective::RefAssociation
       {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   MessageInstanceIsSpecifiedByRequestLinkSet
     all_message_instance_is_specified_by_request_links();
   boolean exists (in MessageInstance instance,
                   in ::UmlCore::Request specification);
   MessageInstanceSet with_specification (
                                    in ::UmlCore::Request specification);
   ::UmlCore::Request with_instance (in MessageInstance instance);
   void add (in MessageInstance instance, in ::UmlCore::Request
       specification)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_instance (in MessageInstance instance,
                         in ::UmlCore::Request specification,
                         in MessageInstance new_instance)
      raises (Reflective::StructuralError,
```

```
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_specification (in MessageInstance instance,
                              in ::UmlCore::Request specification,
                              in ::UmlCore::Request new_specification)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in MessageInstance instance,
                in ::UmlCore::Request specification)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct InstanceReceivesMessageInstanceLink {
   UmlInstance receiver;
   MessageInstance received_message_instance;
};
typedef sequence <InstanceReceivesMessageInstanceLink>
  InstanceReceivesMessageInstanceLinkSet;

interface InstanceReceivesMessageInstance : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   InstanceReceivesMessageInstanceLinkSet
     all_instance_receives_message_instance_links();
   boolean exists (in UmlInstance receiver,
                   in MessageInstance received_message_instance);
   UmlInstance with_received_message_instance (
                           in MessageInstance received_message_instance);
   MessageInstanceSet with_receiver (in UmlInstance receiver);
   void add (in UmlInstance receiver,
             in MessageInstance received_message_instance)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_receiver (in UmlInstance receiver,
                         in MessageInstance received_message_instance,
                         in UmlInstance new_receiver)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_received_message_instance (
```

```
                                 in UmlInstance receiver,
                                 in MessageInstance received_message_instance,
                              in MessageInstance new_received_message_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlInstance receiver,
                  in MessageInstance received_message_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct InstanceOwnsAttributeLinkLink {
   AttributeLink owned_attribute_link;
   UmlInstance owning_instance;
};
typedef sequence <InstanceOwnsAttributeLinkLink>
  InstanceOwnsAttributeLinkLinkSet;

interface InstanceOwnsAttributeLink : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   InstanceOwnsAttributeLinkLinkSet
     all_instance_owns_attribute_link_links();
   boolean exists (in AttributeLink owned_attribute_link,
                   in UmlInstance owning_instance);
  AttributeLinkSet with_owning_instance (in UmlInstance owning_instance);
   UmlInstance with_owned_attribute_link (
                               in AttributeLink owned_attribute_link);
   void add (in AttributeLink owned_attribute_link,
             in UmlInstance owning_instance)
      raises (Reflective::StructuralError, Reflective::SemanticError);
  void add_before_owning_instance (in AttributeLink owned_attribute_link,
                                   in UmlInstance owning_instance,
                                   in UmlInstance before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_owned_attribute_link (
                                 in AttributeLink owned_attribute_link,
                                  in UmlInstance owning_instance,
```

```
                                    in AttributeLink new_owned_attribute_link)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_owning_instance (in AttributeLink owned_attribute_link,
                                 in UmlInstance owning_instance,
                                 in UmlInstance new_owning_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in AttributeLink owned_attribute_link,
                in UmlInstance owning_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct MessageInstanceHasArgumentOfInstanceLink {
   UmlInstance argument;
   MessageInstance argument_owner;
};
typedef sequence <MessageInstanceHasArgumentOfInstanceLink>
  MessageInstanceHasArgumentOfInstanceLinkSet;

interface MessageInstanceHasArgumentOfInstance :
     Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   MessageInstanceHasArgumentOfInstanceLinkSet
     all_message_instance_has_argument_of_instance_links();
   boolean exists (in UmlInstance argument,
                   in MessageInstance argument_owner);
  UmlInstanceSet with_argument_owner (in MessageInstance argument_owner);
   MessageInstanceSet with_argument (in UmlInstance argument);
   void add (in UmlInstance argument, in MessageInstance argument_owner)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_argument (in UmlInstance argument,
                         in MessageInstance argument_owner,
                         in UmlInstance new_argument)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
```

```
    void modify_argument_owner (in UmlInstance argument,
                                in MessageInstance argument_owner,
                                in MessageInstance new_argument_owner)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
  void remove (in UmlInstance argument, in MessageInstance argument_owner)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct ExceptionIsRaisedByBehavioralFeatureLink {
    ::UmlCore::BehavioralFeature uml_context;
    UmlException raised_exception;
};
typedef sequence <ExceptionIsRaisedByBehavioralFeatureLink>
  ExceptionIsRaisedByBehavioralFeatureLinkSet;

interface ExceptionIsRaisedByBehavioralFeature :
      Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
    ExceptionIsRaisedByBehavioralFeatureLinkSet
      all_exception_is_raised_by_behavioral_feature_links();
    boolean exists (in ::UmlCore::BehavioralFeature uml_context,
                    in UmlException raised_exception);
    ::UmlCore::BehavioralFeatureSet with_raised_exception (
                                       in UmlException raised_exception);
    UmlExceptionSet with_uml_context (
                          in ::UmlCore::BehavioralFeature uml_context);
    void add (in ::UmlCore::BehavioralFeature uml_context,
              in UmlException raised_exception)
        raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_uml_context (in ::UmlCore::BehavioralFeature uml_context,
                             in UmlException raised_exception,
                         in ::UmlCore::BehavioralFeature new_uml_context)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_raised_exception (in ::UmlCore::BehavioralFeature
      uml_context,
```

```
                                  in UmlException raised_exception,
                                  in UmlException new_raised_exception)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCore::BehavioralFeature uml_context,
                 in UmlException raised_exception)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct LinkInstantiatesAssociationLink {
   ::UmlCore::Association association;
   Link instance;
};
typedef sequence <LinkInstantiatesAssociationLink>
  LinkInstantiatesAssociationLinkSet;

interface LinkInstantiatesAssociation : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   LinkInstantiatesAssociationLinkSet
     all_link_instantiates_association_links();
   boolean exists (in ::UmlCore::Association association, in Link
     instance);
   ::UmlCore::Association with_instance (in Link instance);
   LinkSet with_association (in ::UmlCore::Association association);
   void add (in ::UmlCore::Association association, in Link instance)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_association (in ::UmlCore::Association association,
                            in Link instance,
                            in ::UmlCore::Association new_association)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_instance (in ::UmlCore::Association association,
                         in Link instance,
                         in Link new_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
```

```
    void remove (in ::UmlCore::Association association, in Link instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct LinkOwnsLinkEndLink {
    Link owning_link;
    LinkEnd link_role;
};
typedef sequence <LinkOwnsLinkEndLink> LinkOwnsLinkEndLinkSet;

interface LinkOwnsLinkEnd : Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
    LinkOwnsLinkEndLinkSet all_link_owns_link_end_links();
    boolean exists (in Link owning_link, in LinkEnd link_role);
    Link with_link_role (in LinkEnd link_role);
    LinkEndSet with_owning_link (in Link owning_link);
    void add (in Link owning_link, in LinkEnd link_role)
      raises (Reflective::StructuralError, Reflective::SemanticError);
    void add_before_owning_link (in Link owning_link,
                                 in LinkEnd link_role,
                                 in Link before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
    void modify_owning_link (in Link owning_link,
                             in LinkEnd link_role,
                             in Link new_owning_link)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
    void modify_link_role (in Link owning_link,
                           in LinkEnd link_role,
                           in LinkEnd new_link_role)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
    void remove (in Link owning_link, in LinkEnd link_role)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
```

```
                Reflective::NotFound);
 };

 struct LinkEndInstantiatesAssociationEndLink {
    ::UmlCore::AssociationEnd association_end;
    LinkEnd instance;
 };
 typedef sequence <LinkEndInstantiatesAssociationEndLink>
   LinkEndInstantiatesAssociationEndLinkSet;

interface LinkEndInstantiatesAssociationEnd : Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
    LinkEndInstantiatesAssociationEndLinkSet
      all_link_end_instantiates_association_end_links();
    boolean exists (in ::UmlCore::AssociationEnd association_end,
                    in LinkEnd instance);
    ::UmlCore::AssociationEnd with_instance (in LinkEnd instance);
    LinkEndSet with_association_end (
                           in ::UmlCore::AssociationEnd association_end);
    void add (in ::UmlCore::AssociationEnd association_end,
              in LinkEnd instance)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_association_end (
                            in ::UmlCore::AssociationEnd association_end,
                            in LinkEnd instance,
                        in ::UmlCore::AssociationEnd new_association_end)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_instance (in ::UmlCore::AssociationEnd association_end,
                          in LinkEnd instance,
                          in LinkEnd new_instance)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in ::UmlCore::AssociationEnd association_end,
                 in LinkEnd instance)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
 };
```

```
struct InstanceSendsMessageInstanceLink {
   UmlInstance sender;
   MessageInstance sent_message_instance;
};
typedef sequence <InstanceSendsMessageInstanceLink>
  InstanceSendsMessageInstanceLinkSet;

interface InstanceSendsMessageInstance : Reflective::RefAssociation {
   readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
   InstanceSendsMessageInstanceLinkSet
     all_instance_sends_message_instance_links();
   boolean exists (in UmlInstance sender,
                   in MessageInstance sent_message_instance);
   UmlInstance with_sent_message_instance (
                                  in MessageInstance sent_message_instance);
   MessageInstanceSet with_sender (in UmlInstance sender);
   void add (in UmlInstance sender, in MessageInstance
     sent_message_instance)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_sender (in UmlInstance sender,
                       in MessageInstance sent_message_instance,
                       in UmlInstance new_sender)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_sent_message_instance (
                                   in UmlInstance sender,
                                  in MessageInstance sent_message_instance,
                              in MessageInstance new_sent_message_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlInstance sender,
                in MessageInstance sent_message_instance)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ActionSequenceOwnsActionLink {
```

```
    UmlCommonBehavior::ActionSequence action_sequence;
    UmlCommonBehavior::Action action;
};
typedef sequence <ActionSequenceOwnsActionLink>
  ActionSequenceOwnsActionLinkSet;

interface ActionSequenceOwnsAction : Reflective::RefAssociation {
    readonly attribute UmlCommonBehaviorPackage enclosing_package_ref;
    ActionSequenceOwnsActionLinkSet
      all_action_sequence_owns_action_links();
    boolean exists (in UmlCommonBehavior::ActionSequence action_sequence,
                    in UmlCommonBehavior::Action action);
    UmlCommonBehavior::ActionSequence with_action (
                                    in UmlCommonBehavior::Action action);
    UmlCommonBehavior::ActionSet with_action_sequence (
                   in UmlCommonBehavior::ActionSequence action_sequence);
    void add (in UmlCommonBehavior::ActionSequence action_sequence,
              in UmlCommonBehavior::Action action)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_action_sequence (
                    in UmlCommonBehavior::ActionSequence action_sequence,
                     in UmlCommonBehavior::Action action,
               in UmlCommonBehavior::ActionSequence new_action_sequence)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void modify_action (in UmlCommonBehavior::ActionSequence
      action_sequence,
                         in UmlCommonBehavior::Action action,
                         in UmlCommonBehavior::Action new_action)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
    void remove (in UmlCommonBehavior::ActionSequence action_sequence,
                 in UmlCommonBehavior::Action action)
       raises (Reflective::StructuralError,
               Reflective::SemanticError,
               Reflective::NotFound);
};

interface UmlCommonBehaviorPackageFactory  {
```

```
    UmlCommonBehaviorPackage create_uml_common_behavior_package ()
       raises (Reflective::SemanticError);
};

interface UmlCommonBehaviorPackage : Reflective::RefPackage {
    readonly attribute CallClass call_class_ref;
    readonly attribute LinkEndClass link_end_class_ref;
    readonly attribute LinkClass link_class_ref;
    readonly attribute AttributeLinkClass attribute_link_class_ref;
    readonly attribute UmlInstanceClass uml_instance_class_ref;
    readonly attribute UmlObjectClass uml_object_class_ref;
    readonly attribute MessageInstanceClass message_instance_class_ref;
    readonly attribute DataValueClass data_value_class_ref;
    readonly attribute SignalClass signal_class_ref;
    readonly attribute UmlExceptionClass uml_exception_class_ref;
    readonly attribute ReceptionClass reception_class_ref;
    readonly attribute ArgumentClass argument_class_ref;
    readonly attribute ActionClass action_class_ref;
    readonly attribute CallActionClass call_action_class_ref;
    readonly attribute CreateActionClass create_action_class_ref;
    readonly attribute DestroyActionClass destroy_action_class_ref;
    readonly attribute LocalInvocationClass local_invocation_class_ref;
    readonly attribute SendActionClass send_action_class_ref;
    readonly attribute ReturnActionClass return_action_class_ref;
    readonly attribute TerminateActionClass terminate_action_class_ref;
    readonly attribute UninterpretedActionClass
      uninterpreted_action_class_ref;
    readonly attribute ActionSequenceClass action_sequence_class_ref;
    readonly attribute LinkObjectClass link_object_class_ref;

    readonly attribute InstanceInstantiatesClassifier
      instance_instantiates_classifier_ref;
    readonly attribute ActionOwnsArgument action_owns_argument_ref;
    readonly attribute CreateActionInstantiatesClassifier
      create_action_instantiates_classifier_ref;
    readonly attribute AttributeLinkIsInstanceOfAttribute
      attribute_link_is_instance_of_attribute_ref;
    readonly attribute AttributeLilnkHasValueOfLink
      attribute_lilnk_has_value_of_link_ref;
    readonly attribute LinkEndIsOfTypeInstance
      link_end_is_of_type_instance_ref;
```

```
      readonly attribute ReceptionFeatureReceivesSignal
        reception_feature_receives_signal_ref;
      readonly attribute SignalOwnsParameter signal_owns_parameter_ref;
      readonly attribute ActionIsInitiatedByRequest
        action_is_initiated_by_request_ref;
      readonly attribute MessageInstanceIsSpecifiedByRequest
        message_instance_is_specified_by_request_ref;
      readonly attribute InstanceReceivesMessageInstance
        instance_receives_message_instance_ref;
      readonly attribute InstanceOwnsAttributeLink
        instance_owns_attribute_link_ref;
      readonly attribute MessageInstanceHasArgumentOfInstance
        message_instance_has_argument_of_instance_ref;
      readonly attribute ExceptionIsRaisedByBehavioralFeature
        exception_is_raised_by_behavioral_feature_ref;
      readonly attribute LinkInstantiatesAssociation
        link_instantiates_association_ref;
      readonly attribute LinkOwnsLinkEnd link_owns_link_end_ref;
      readonly attribute LinkEndInstantiatesAssociationEnd
        link_end_instantiates_association_end_ref;
      readonly attribute InstanceSendsMessageInstance
        instance_sends_message_instance_ref;
      readonly attribute ActionSequenceOwnsAction
        action_sequence_owns_action_ref;
   };
};
```

5.4.6 UMLStateMachines

```
#include "UmlCommonBehavior.idl"

module UmlStateMachines {
   interface UmlStateMachinesPackage;
   interface ActivityState;
   interface ActivityStateClass;
   typedef sequence<ActivityState> ActivityStateUList;
   interface CompositeState;
   interface CompositeStateClass;
   typedef sequence<CompositeState> CompositeStateUList;
   interface TimeEvent;
   interface TimeEventClass;
```

```
typedef sequence<TimeEvent> TimeEventUList;
interface ActionState;
interface ActionStateClass;
typedef sequence<ActionState> ActionStateUList;
interface CallEvent;
interface CallEventClass;
typedef sequence<CallEvent> CallEventUList;
typedef sequence<CallEvent> CallEventSet;
interface ChangeEvent;
interface ChangeEventClass;
typedef sequence<ChangeEvent> ChangeEventUList;
interface SignalEvent;
interface SignalEventClass;
typedef sequence<SignalEvent> SignalEventUList;
typedef sequence<SignalEvent> SignalEventSet;
interface Pseudostate;
interface PseudostateClass;
typedef sequence<Pseudostate> PseudostateUList;
interface Event;
interface EventClass;
typedef sequence<Event> EventUList;
typedef sequence<Event> EventSet;
interface StateVertex;
interface StateVertexClass;
typedef sequence<StateVertex> StateVertexUList;
typedef sequence<StateVertex> StateVertexSet;
interface StateMachine;
interface StateMachineClass;
typedef sequence<StateMachine> StateMachineUList;
typedef sequence<StateMachine> StateMachineSet;
interface SimpleState;
interface SimpleStateClass;
typedef sequence<SimpleState> SimpleStateUList;
interface ObjectFlowState;
interface ObjectFlowStateClass;
typedef sequence<ObjectFlowState> ObjectFlowStateUList;
typedef sequence<ObjectFlowState> ObjectFlowStateSet;
interface SubmachineState;
interface SubmachineStateClass;
typedef sequence<SubmachineState> SubmachineStateUList;
typedef sequence<SubmachineState> SubmachineStateSet;
```

```
interface ActivityModel;
interface ActivityModelClass;
typedef sequence<ActivityModel> ActivityModelUList;
interface Guard;
interface GuardClass;
typedef sequence<Guard> GuardUList;
interface Transition;
interface TransitionClass;
typedef sequence<Transition> TransitionUList;
typedef sequence<Transition> TransitionSet;
interface ClassifierInState;
interface ClassifierInStateClass;
typedef sequence<ClassifierInState> ClassifierInStateUList;
typedef sequence<ClassifierInState> ClassifierInStateSet;
interface Partition;
interface PartitionClass;
typedef sequence<Partition> PartitionUList;
typedef sequence<Partition> PartitionSet;
interface State;
interface StateClass;
typedef sequence<State> StateUList;
typedef sequence<State> StateSet;

interface EventClass : ::UmlCore::ModelElementClass {
   readonly attribute EventUList all_of_kind_event;
};

interface Event : EventClass, ::UmlCore::ModelElement {
   UmlStateMachines::StateSet state ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_state (in UmlStateMachines::StateSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_state ()
      raises (Reflective::SemanticError);
   UmlStateMachines::TransitionSet transition ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_transition (in UmlStateMachines::TransitionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_transition ()
      raises (Reflective::SemanticError);
};
```

```

interface CallEventClass : EventClass {
   readonly attribute CallEventUList all_of_kind_call_event;
   readonly attribute CallEventUList all_of_type_call_event;
   CallEvent create_call_event (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
 };

 interface CallEvent : CallEventClass, Event {
   ::UmlCore::Operation operation ()
      raises (Reflective::SemanticError);
   void set_operation (in ::UmlCore::Operation new_value)
      raises (Reflective::SemanticError);
 };

 interface ChangeEventClass : EventClass {
   readonly attribute ChangeEventUList all_of_kind_change_event;
   readonly attribute ChangeEventUList all_of_type_change_event;
   ChangeEvent create_change_event (
                       in ::UmlCore::Name name,
                    in ::UmlCore::BooleanExpression change_expression)
      raises (Reflective::SemanticError);
 };

 interface ChangeEvent : ChangeEventClass, Event {
   ::UmlCore::BooleanExpression change_expression ()
      raises (Reflective::SemanticError);
   void set_change_expression (in ::UmlCore::BooleanExpression new_value)
      raises (Reflective::SemanticError);
 };

 interface SignalEventClass : EventClass {
   readonly attribute SignalEventUList all_of_kind_signal_event;
   readonly attribute SignalEventUList all_of_type_signal_event;
   SignalEvent create_signal_event (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
 };

 interface SignalEvent : SignalEventClass, Event {
   ::UmlCommonBehavior::Signal signal ()
      raises (Reflective::SemanticError);
```

```
   void set_signal (in ::UmlCommonBehavior::Signal new_value)
      raises (Reflective::SemanticError);
};

interface TimeEventClass : EventClass {
   readonly attribute TimeEventUList all_of_kind_time_event;
   readonly attribute TimeEventUList all_of_type_time_event;
   TimeEvent create_time_event (in ::UmlCore::Name name,
                             in ::UmlCore::TimeExpression duration)
      raises (Reflective::SemanticError);
};

interface TimeEvent : TimeEventClass, Event {
   ::UmlCore::TimeExpression duration ()
      raises (Reflective::SemanticError);
   void set_duration (in ::UmlCore::TimeExpression new_value)
      raises (Reflective::SemanticError);
};

interface StateVertexClass : ::UmlCore::ModelElementClass {
   readonly attribute StateVertexUList all_of_kind_state_vertex;
};

interface StateVertex : StateVertexClass, ::UmlCore::ModelElement {
   CompositeState parent ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_parent (in CompositeState new_value)
      raises (Reflective::SemanticError);
   void unset_parent ()
      raises (Reflective::SemanticError);
   TransitionSet outgoing ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_outgoing (in TransitionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_outgoing ()
      raises (Reflective::SemanticError);
   TransitionSet incoming ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_incoming (in TransitionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_incoming ()
```

```
      raises (Reflective::SemanticError);
};

interface GuardClass : ::UmlCore::ModelElementClass {
   readonly attribute GuardUList all_of_kind_guard;
   readonly attribute GuardUList all_of_type_guard;
   Guard create_guard (in ::UmlCore::Name name,
                       in ::UmlCore::BooleanExpression expression)
      raises (Reflective::SemanticError);
};

interface Guard : GuardClass, ::UmlCore::ModelElement {
   ::UmlCore::BooleanExpression expression ()
      raises (Reflective::SemanticError);
   void set_expression (in ::UmlCore::BooleanExpression new_value)
      raises (Reflective::SemanticError);
   UmlStateMachines::Transition transition ()
      raises (Reflective::SemanticError);
   void set_transition (in UmlStateMachines::Transition new_value)
      raises (Reflective::SemanticError);
};

interface TransitionClass : ::UmlCore::ModelElementClass {
   readonly attribute TransitionUList all_of_kind_transition;
   readonly attribute TransitionUList all_of_type_transition;
   Transition create_transition (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface Transition : TransitionClass, ::UmlCore::ModelElement {
   UmlStateMachines::Guard guard ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_guard (in UmlStateMachines::Guard new_value)
      raises (Reflective::SemanticError);
   void add_guard_before (in UmlStateMachines::Guard new_value,
                          in UmlStateMachines::Guard before)
      raises (Reflective::StructuralError,
              Reflective::NotFound,
              Reflective::SemanticError);
   void unset_guard ()
      raises (Reflective::SemanticError);
```

```
   ::UmlCommonBehavior::ActionSequence effect ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_effect (in ::UmlCommonBehavior::ActionSequence new_value)
      raises (Reflective::SemanticError);
   void unset_effect ()
      raises (Reflective::SemanticError);
   UmlStateMachines::State state ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_state (in UmlStateMachines::State new_value)
      raises (Reflective::SemanticError);
   void unset_state ()
      raises (Reflective::SemanticError);
   Event trigger ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_trigger (in Event new_value)
      raises (Reflective::SemanticError);
   void unset_trigger ()
      raises (Reflective::SemanticError);
   UmlStateMachines::StateMachine state_machine ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_state_machine (in UmlStateMachines::StateMachine new_value)
      raises (Reflective::SemanticError);
   void unset_state_machine ()
      raises (Reflective::SemanticError);
   StateVertex source ()
      raises (Reflective::SemanticError);
   void set_source (in StateVertex new_value)
      raises (Reflective::SemanticError);
   StateVertex target ()
      raises (Reflective::SemanticError);
   void set_target (in StateVertex new_value)
      raises (Reflective::SemanticError);
};

interface PseudostateClass : StateVertexClass {
   readonly attribute PseudostateUList all_of_kind_pseudostate;
   readonly attribute PseudostateUList all_of_type_pseudostate;
   Pseudostate create_pseudostate (in ::UmlCore::Name name,
                                   in ::UmlCore::PseudostateKind kind)
      raises (Reflective::SemanticError);
};
```

```
interface Pseudostate : PseudostateClass, StateVertex {
   ::UmlCore::PseudostateKind kind ()
      raises (Reflective::SemanticError);
   void set_kind (in ::UmlCore::PseudostateKind new_value)
      raises (Reflective::SemanticError);
};

interface StateClass : StateVertexClass {
   readonly attribute StateUList all_of_kind_state;
   readonly attribute StateUList all_of_type_state;
   State create_state (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface State : StateClass, StateVertex {
   ::UmlCommonBehavior::ActionSequence entry ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_entry (in ::UmlCommonBehavior::ActionSequence new_value)
      raises (Reflective::SemanticError);
   void unset_entry ()
      raises (Reflective::SemanticError);
   ::UmlCommonBehavior::ActionSequence exit ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_exit (in ::UmlCommonBehavior::ActionSequence new_value)
      raises (Reflective::SemanticError);
   void unset_exit ()
      raises (Reflective::SemanticError);
   UmlStateMachines::ClassifierInStateSet classifier_in_state ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_classifier_in_state (
                  in UmlStateMachines::ClassifierInStateSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_classifier_in_state ()
      raises (Reflective::SemanticError);
   UmlStateMachines::StateMachine state_machine ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_state_machine (in UmlStateMachines::StateMachine new_value)
      raises (Reflective::SemanticError);
   void unset_state_machine ()
      raises (Reflective::SemanticError);
```

```
      EventSet deferred_event ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_deferred_event (in EventSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_deferred_event ()
         raises (Reflective::SemanticError);
      TransitionSet internal_transition ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_internal_transition (in TransitionSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_internal_transition ()
         raises (Reflective::SemanticError);
   };

   interface CompositeStateClass : StateClass {
      readonly attribute CompositeStateUList all_of_kind_composite_state;
      readonly attribute CompositeStateUList all_of_type_composite_state;
      CompositeState create_composite_state (in ::UmlCore::Name name,
                                       in boolean is_concurent)
         raises (Reflective::SemanticError);
   };

   interface CompositeState : CompositeStateClass, State {
      boolean is_concurent ()
         raises (Reflective::SemanticError);
      void set_is_concurent (in boolean new_value)
         raises (Reflective::SemanticError);
      StateVertexSet substate ()
         raises (Reflective::SemanticError);
      void add_substate (in StateVertexSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_substate ()
         raises (Reflective::SemanticError);
   };

   interface PartitionClass : ::UmlCore::ModelElementClass {
      readonly attribute PartitionUList all_of_kind_partition;
      readonly attribute PartitionUList all_of_type_partition;
      Partition create_partition (in ::UmlCore::Name name)
         raises (Reflective::SemanticError);
   };
```

```
   interface Partition : PartitionClass, ::UmlCore::ModelElement {
      UmlStateMachines::ActivityModel activity_model ()
         raises (Reflective::SemanticError);
      void set_activity_model (in UmlStateMachines::ActivityModel new_value)
         raises (Reflective::SemanticError);
      ::UmlCore::ModelElementSet contents ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_contents (in ::UmlCore::ModelElementSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_contents ()
         raises (Reflective::SemanticError);
   };

   interface ClassifierInStateClass : ::UmlCore::ClassifierClass {
      readonly attribute ClassifierInStateUList
         all_of_kind_classifier_in_state;
      readonly attribute ClassifierInStateUList
         all_of_type_classifier_in_state;
     ClassifierInState create_classifier_in_state (in ::UmlCore::Name name,
                                                   in boolean is_root,
                                                   in boolean is_leaf,
                                                   in boolean is_abstract)
         raises (Reflective::SemanticError);
   };

  interface ClassifierInState : ClassifierInStateClass, ::UmlCore::Classifier
         {
      State in_state ()
         raises (Reflective::SemanticError);
      void set_in_state (in State new_value)
         raises (Reflective::SemanticError);
      UmlStateMachines::ObjectFlowStateSet object_flow_state ()
         raises (Reflective::NotSet, Reflective::SemanticError);
      void add_object_flow_state (
                           in UmlStateMachines::ObjectFlowStateSet new_value)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void remove_object_flow_state ()
         raises (Reflective::SemanticError);
      ::UmlCore::Classifier type ()
         raises (Reflective::SemanticError);
```

```
   void set_type (in ::UmlCore::Classifier new_value)
      raises (Reflective::SemanticError);
};

interface StateMachineClass : ::UmlCore::ModelElementClass {
   readonly attribute StateMachineUList all_of_kind_state_machine;
   readonly attribute StateMachineUList all_of_type_state_machine;
   StateMachine create_state_machine (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface StateMachine : StateMachineClass, ::UmlCore::ModelElement {
   State top ()
      raises (Reflective::SemanticError);
   void set_top (in State new_value)
      raises (Reflective::SemanticError);
   TransitionSet transitions ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_transitions (in TransitionSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_transitions ()
      raises (Reflective::SemanticError);
   UmlStateMachines::SubmachineStateSet submachine_state ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_submachine_state (
                    in UmlStateMachines::SubmachineStateSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_submachine_state ()
      raises (Reflective::SemanticError);
   ::UmlCore::ModelElement uml_context ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_uml_context (in ::UmlCore::ModelElement new_value)
      raises (Reflective::SemanticError);
   void unset_uml_context ()
      raises (Reflective::SemanticError);
};

interface ActivityModelClass : StateMachineClass {
   readonly attribute ActivityModelUList all_of_kind_activity_model;
   readonly attribute ActivityModelUList all_of_type_activity_model;
   ActivityModel create_activity_model (in ::UmlCore::Name name)
```

```
        raises (Reflective::SemanticError);
};

interface ActivityModel : ActivityModelClass, StateMachine {
    PartitionSet owned_partition ()
        raises (Reflective::NotSet, Reflective::SemanticError);
    void add_owned_partition (in PartitionSet new_value)
        raises (Reflective::StructuralError, Reflective::SemanticError);
    void remove_owned_partition ()
        raises (Reflective::SemanticError);
};

interface SimpleStateClass : StateClass {
    readonly attribute SimpleStateUList all_of_kind_simple_state;
    readonly attribute SimpleStateUList all_of_type_simple_state;
    SimpleState create_simple_state (in ::UmlCore::Name name)
        raises (Reflective::SemanticError);
};

interface SimpleState : SimpleStateClass, State { };

interface ActivityStateClass : SimpleStateClass {
    readonly attribute ActivityStateUList all_of_kind_activity_state;
    readonly attribute ActivityStateUList all_of_type_activity_state;
    ActivityState create_activity_state (in ::UmlCore::Name name)
        raises (Reflective::SemanticError);
};

interface ActivityState : ActivityStateClass, SimpleState { };

interface ObjectFlowStateClass : SimpleStateClass {
    readonly attribute ObjectFlowStateUList all_of_kind_object_flow_state;
    readonly attribute ObjectFlowStateUList all_of_type_object_flow_state;
    ObjectFlowState create_object_flow_state (in ::UmlCore::Name name)
        raises (Reflective::SemanticError);
};

interface ObjectFlowState : ObjectFlowStateClass, SimpleState {
    UmlStateMachines::ClassifierInState type_state ()
        raises (Reflective::SemanticError);
    void set_type_state (in UmlStateMachines::ClassifierInState new_value)
```

```
      raises (Reflective::SemanticError);
};

interface ActionStateClass : SimpleStateClass {
   readonly attribute ActionStateUList all_of_kind_action_state;
   readonly attribute ActionStateUList all_of_type_action_state;
   ActionState create_action_state (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface ActionState : ActionStateClass, SimpleState { };

interface SubmachineStateClass : StateClass {
   readonly attribute SubmachineStateUList all_of_kind_submachine_state;
   readonly attribute SubmachineStateUList all_of_type_submachine_state;
   SubmachineState create_submachine_state (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface SubmachineState : SubmachineStateClass, State {
   UmlStateMachines::StateMachine submachine ()
      raises (Reflective::SemanticError);
   void set_submachine (in UmlStateMachines::StateMachine new_value)
      raises (Reflective::SemanticError);
};

struct StateOwnsEntryActionSequenceLink {
   State entry_action_state;
   ::UmlCommonBehavior::ActionSequence entry;
};
typedef sequence <StateOwnsEntryActionSequenceLink>
  StateOwnsEntryActionSequenceLinkSet;

interface StateOwnsEntryActionSequence : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateOwnsEntryActionSequenceLinkSet
     all_state_owns_entry_action_sequence_links();
   boolean exists (in State entry_action_state,
                   in ::UmlCommonBehavior::ActionSequence entry);
   State with_entry (in ::UmlCommonBehavior::ActionSequence entry);
   ::UmlCommonBehavior::ActionSequence with_entry_action_state (
```

```
                                                              in State entry_action_state);
   void add (in State entry_action_state,
             in ::UmlCommonBehavior::ActionSequence entry)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_entry_action_state (
                                   in State entry_action_state,
                                in ::UmlCommonBehavior::ActionSequence entry,
                                   in State new_entry_action_state)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_entry (in State entry_action_state,
                      in ::UmlCommonBehavior::ActionSequence entry,
                      in ::UmlCommonBehavior::ActionSequence new_entry)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in State entry_action_state,
                in ::UmlCommonBehavior::ActionSequence entry)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct StateOwnsExitActionSequenceLink {
   State exit_action_state;
   ::UmlCommonBehavior::ActionSequence exit;
};
typedef sequence <StateOwnsExitActionSequenceLink>
StateOwnsExitActionSequenceLinkSet;

interface StateOwnsExitActionSequence : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateOwnsExitActionSequenceLinkSet
     all_state_owns_exit_action_sequence_links();
   boolean exists (in State exit_action_state,
                   in ::UmlCommonBehavior::ActionSequence exit);
   State with_exit (in ::UmlCommonBehavior::ActionSequence exit);
   ::UmlCommonBehavior::ActionSequence with_exit_action_state (
                                                               in State exit_action_state);
   void add (in State exit_action_state,
```

```
                in ::UmlCommonBehavior::ActionSequence exit)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_exit_action_state (in State exit_action_state,
                                     in ::UmlCommonBehavior::ActionSequence exit,
                                          in State new_exit_action_state)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void modify_exit (in State exit_action_state,
                        in ::UmlCommonBehavior::ActionSequence exit,
                        in ::UmlCommonBehavior::ActionSequence new_exit)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in State exit_action_state,
                   in ::UmlCommonBehavior::ActionSequence exit)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   struct TransitionOwnsGuardLink {
      UmlStateMachines::Guard guard;
      UmlStateMachines::Transition transition;
   };
   typedef sequence <TransitionOwnsGuardLink> TransitionOwnsGuardLinkSet;

   interface TransitionOwnsGuard : Reflective::RefAssociation {
      readonly attribute UmlStateMachinesPackage enclosing_package_ref;
      TransitionOwnsGuardLinkSet all_transition_owns_guard_links();
      boolean exists (in UmlStateMachines::Guard guard,
                      in UmlStateMachines::Transition transition);
      UmlStateMachines::Guard with_transition (
                                    in UmlStateMachines::Transition transition);
      UmlStateMachines::Transition with_guard (in UmlStateMachines::Guard
         guard);
      void add (in UmlStateMachines::Guard guard,
                in UmlStateMachines::Transition transition)
         raises (Reflective::StructuralError, Reflective::SemanticError);
      void add_before_guard (in UmlStateMachines::Guard guard,
                             in UmlStateMachines::Transition transition,
```

```
                              in UmlStateMachines::Guard before)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_guard (in UmlStateMachines::Guard guard,
                      in UmlStateMachines::Transition transition,
                      in UmlStateMachines::Guard new_guard)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_transition (in UmlStateMachines::Guard guard,
                           in UmlStateMachines::Transition transition,
                           in UmlStateMachines::Transition new_transition)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlStateMachines::Guard guard,
                in UmlStateMachines::Transition transition)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct SignalEventIsOccurrenceOfSignalLink {
   ::UmlCommonBehavior::Signal signal;
   SignalEvent occurrence;
};
typedef sequence <SignalEventIsOccurrenceOfSignalLink>
  SignalEventIsOccurrenceOfSignalLinkSet;

interface SignalEventIsOccurrenceOfSignal : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   SignalEventIsOccurrenceOfSignalLinkSet
     all_signal_event_is_occurrence_of_signal_links();
   boolean exists (in ::UmlCommonBehavior::Signal signal,
                   in SignalEvent occurrence);
  ::UmlCommonBehavior::Signal with_occurrence (in SignalEvent occurrence);
   SignalEventSet with_signal (in ::UmlCommonBehavior::Signal signal);
   void add (in ::UmlCommonBehavior::Signal signal, in SignalEvent
      occurrence)
      raises (Reflective::StructuralError, Reflective::SemanticError);
```

```
   void modify_signal (in ::UmlCommonBehavior::Signal signal,
                       in SignalEvent occurrence,
                       in ::UmlCommonBehavior::Signal new_signal)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_occurrence (in ::UmlCommonBehavior::Signal signal,
                           in SignalEvent occurrence,
                           in SignalEvent new_occurrence)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCommonBehavior::Signal signal,
                in SignalEvent occurrence)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct ActivityModelOwnsPartitionLink {
   UmlStateMachines::ActivityModel activity_model;
   Partition owned_partition;
};
typedef sequence <ActivityModelOwnsPartitionLink>
ActivityModelOwnsPartitionLinkSet;

interface ActivityModelOwnsPartition : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   ActivityModelOwnsPartitionLinkSet
     all_activity_model_owns_partition_links();
   boolean exists (in UmlStateMachines::ActivityModel activity_model,
                   in Partition owned_partition);
   UmlStateMachines::ActivityModel with_owned_partition (
                                            in Partition owned_partition);
   PartitionSet with_activity_model (
                      in UmlStateMachines::ActivityModel activity_model);
   void add (in UmlStateMachines::ActivityModel activity_model,
             in Partition owned_partition)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_activity_model (
                       in UmlStateMachines::ActivityModel activity_model,
```

```
                              in Partition owned_partition,
                          in UmlStateMachines::ActivityModel new_activity_model)
            raises (Reflective::StructuralError,
                    Reflective::SemanticError,
                    Reflective::NotFound);
        void modify_owned_partition (
                        in UmlStateMachines::ActivityModel activity_model,
                           in Partition owned_partition,
                           in Partition new_owned_partition)
            raises (Reflective::StructuralError,
                    Reflective::SemanticError,
                    Reflective::NotFound);
        void remove (in UmlStateMachines::ActivityModel activity_model,
                      in Partition owned_partition)
            raises (Reflective::StructuralError,
                    Reflective::SemanticError,
                    Reflective::NotFound);
    };

    struct PartitionHasContextOfModelElementLink {
        ::UmlCore::ModelElement contents;
        Partition context_partition;
    };
    typedef sequence <PartitionHasContextOfModelElementLink>
      PartitionHasContextOfModelElementLinkSet;

    interface PartitionHasContextOfModelElement : Reflective::RefAssociation {
        readonly attribute UmlStateMachinesPackage enclosing_package_ref;
        PartitionHasContextOfModelElementLinkSet
          all_partition_has_context_of_model_element_links();
        boolean exists (in ::UmlCore::ModelElement contents,
                        in Partition context_partition);
        ::UmlCore::ModelElementSet with_context_partition (
                                                in Partition context_partition);
        PartitionSet with_contents (in ::UmlCore::ModelElement contents);
        void add (in ::UmlCore::ModelElement contents,
                  in Partition context_partition)
            raises (Reflective::StructuralError, Reflective::SemanticError);
        void modify_contents (in ::UmlCore::ModelElement contents,
                              in Partition context_partition,
                              in ::UmlCore::ModelElement new_contents)
```

```
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_context_partition (in ::UmlCore::ModelElement contents,
                                   in Partition context_partition,
                                   in Partition new_context_partition)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void remove (in ::UmlCore::ModelElement contents,
                  in Partition context_partition)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct ClassifierInStateIsInStateStateLink {
    UmlStateMachines::ClassifierInState classifier_in_state;
    State in_state;
};
typedef sequence <ClassifierInStateIsInStateStateLink>
  ClassifierInStateIsInStateStateLinkSet;

interface ClassifierInStateIsInStateState : Reflective::RefAssociation {
    readonly attribute UmlStateMachinesPackage enclosing_package_ref;
    ClassifierInStateIsInStateStateLinkSet
      all_classifier_in_state_is_in_state_state_links();
    boolean exists (in UmlStateMachines::ClassifierInState
        classifier_in_state,
                    in State in_state);
    UmlStateMachines::ClassifierInStateSet with_in_state (in State
        in_state);
    State with_classifier_in_state (
               in UmlStateMachines::ClassifierInState classifier_in_state);
    void add (in UmlStateMachines::ClassifierInState classifier_in_state,
              in State in_state)
        raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_classifier_in_state (
                in UmlStateMachines::ClassifierInState classifier_in_state,
                in State in_state,
            in UmlStateMachines::ClassifierInState new_classifier_in_state)
```

```
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_in_state (
                in UmlStateMachines::ClassifierInState classifier_in_state,
                    in State in_state,
                    in State new_in_state)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
   void remove (in UmlStateMachines::ClassifierInState classifier_in_state,
                  in State in_state)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
};

struct ObjectFlowStateRepresentsClassifierInStateLink {
   ClassifierInState type_state;
   UmlStateMachines::ObjectFlowState object_flow_state;
};
typedef sequence <ObjectFlowStateRepresentsClassifierInStateLink>
  ObjectFlowStateRepresentsClassifierInStateLinkSet;

interface ObjectFlowStateRepresentsClassifierInState :
                                            Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   ObjectFlowStateRepresentsClassifierInStateLinkSet
     all_object_flow_state_represents_classifier_in_state_links();
   boolean exists (in ClassifierInState type_state,
                  in UmlStateMachines::ObjectFlowState object_flow_state);
   ClassifierInState with_object_flow_state (
                  in UmlStateMachines::ObjectFlowState object_flow_state);
   UmlStateMachines::ObjectFlowStateSet with_type_state (
                                        in ClassifierInState type_state);
   void add (in ClassifierInState type_state,
             in UmlStateMachines::ObjectFlowState object_flow_state)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_type_state (
                      in ClassifierInState type_state,
                   in UmlStateMachines::ObjectFlowState object_flow_state,
```

```
                          in ClassifierInState new_type_state)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
     void modify_object_flow_state (
                      in ClassifierInState type_state,
                      in UmlStateMachines::ObjectFlowState object_flow_state,
                  in UmlStateMachines::ObjectFlowState new_object_flow_state)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
     void remove (in ClassifierInState type_state,
                  in UmlStateMachines::ObjectFlowState object_flow_state)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
  };

  struct ClassifierInStateIsOfTypeClassifierLink {
     ::UmlCore::Classifier type;
     UmlStateMachines::ClassifierInState classifier_in_state;
  };
  typedef sequence <ClassifierInStateIsOfTypeClassifierLink>
    ClassifierInStateIsOfTypeClassifierLinkSet;

 interface ClassifierInStateIsOfTypeClassifier : Reflective::RefAssociation
        {
     readonly attribute UmlStateMachinesPackage enclosing_package_ref;
     ClassifierInStateIsOfTypeClassifierLinkSet
       all_classifier_in_state_is_of_type_classifier_links();
     boolean exists (
                     in ::UmlCore::Classifier type,
                 in UmlStateMachines::ClassifierInState classifier_in_state);
     ::UmlCore::Classifier with_classifier_in_state (
                 in UmlStateMachines::ClassifierInState classifier_in_state);
     UmlStateMachines::ClassifierInStateSet with_type (
                                    in ::UmlCore::Classifier type);
     void add (in ::UmlCore::Classifier type,
               in UmlStateMachines::ClassifierInState classifier_in_state)
        raises (Reflective::StructuralError, Reflective::SemanticError);
     void modify_type (
```

```
                  in ::UmlCore::Classifier type,
              in UmlStateMachines::ClassifierInState classifier_in_state,
                  in ::UmlCore::Classifier new_type)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_classifier_in_state (
              in ::UmlCore::Classifier type,
              in UmlStateMachines::ClassifierInState classifier_in_state,
          in UmlStateMachines::ClassifierInState new_classifier_in_state)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in ::UmlCore::Classifier type,
              in UmlStateMachines::ClassifierInState classifier_in_state)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct StateMachineOwnsTopStateLink {
   State top;
   UmlStateMachines::StateMachine state_machine;
};
typedef sequence <StateMachineOwnsTopStateLink>
  StateMachineOwnsTopStateLinkSet;

interface StateMachineOwnsTopState : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateMachineOwnsTopStateLinkSet
      all_state_machine_owns_top_state_links();
   boolean exists (in State top,
                   in UmlStateMachines::StateMachine state_machine);
   State with_state_machine (in UmlStateMachines::StateMachine
      state_machine);
   UmlStateMachines::StateMachine with_top (in State top);
  void add (in State top, in UmlStateMachines::StateMachine state_machine)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_top (in State top,
                    in UmlStateMachines::StateMachine state_machine,
                    in State new_top)
```

```
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_state_machine (
                         in State top,
                         in UmlStateMachines::StateMachine state_machine,
                     in UmlStateMachines::StateMachine new_state_machine)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in State top, in UmlStateMachines::StateMachine
      state_machine)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct StateDefersEventLink {
   UmlStateMachines::State state;
   Event deferred_event;
};
typedef sequence <StateDefersEventLink> StateDefersEventLinkSet;

interface StateDefersEvent : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateDefersEventLinkSet all_state_defers_event_links();
   boolean exists (in UmlStateMachines::State state, in Event
      deferred_event);
   UmlStateMachines::StateSet with_deferred_event (in Event
      deferred_event);
   EventSet with_state (in UmlStateMachines::State state);
   void add (in UmlStateMachines::State state, in Event deferred_event)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_state (in UmlStateMachines::State state,
                      in Event deferred_event,
                      in UmlStateMachines::State new_state)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_deferred_event (in UmlStateMachines::State state,
                               in Event deferred_event,
```

```
                                    in Event new_deferred_event)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
      void remove (in UmlStateMachines::State state, in Event deferred_event)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
   };

   struct CallEventIsOccurrenceOfOperationLink {
      CallEvent occurrence;
      ::UmlCore::Operation operation;
   };
   typedef sequence <CallEventIsOccurrenceOfOperationLink>
     CallEventIsOccurrenceOfOperationLinkSet;

   interface CallEventIsOccurrenceOfOperation : Reflective::RefAssociation {
      readonly attribute UmlStateMachinesPackage enclosing_package_ref;
      CallEventIsOccurrenceOfOperationLinkSet
        all_call_event_is_occurrence_of_operation_links();
      boolean exists (in CallEvent occurrence,
                      in ::UmlCore::Operation operation);
      CallEventSet with_operation (in ::UmlCore::Operation operation);
      ::UmlCore::Operation with_occurrence (in CallEvent occurrence);
      void add (in CallEvent occurrence, in ::UmlCore::Operation operation)
          raises (Reflective::StructuralError, Reflective::SemanticError);
      void modify_occurrence (in CallEvent occurrence,
                              in ::UmlCore::Operation operation,
                              in CallEvent new_occurrence)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
      void modify_operation (in CallEvent occurrence,
                             in ::UmlCore::Operation operation,
                             in ::UmlCore::Operation new_operation)
          raises (Reflective::StructuralError,
                  Reflective::SemanticError,
                  Reflective::NotFound);
     void remove (in CallEvent occurrence, in ::UmlCore::Operation operation)
          raises (Reflective::StructuralError,
```

```
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct CompositeStateOwnsSubstateStateVertexLink {
   CompositeState parent;
   StateVertex substate;
};
typedef sequence <CompositeStateOwnsSubstateStateVertexLink>
  CompositeStateOwnsSubstateStateVertexLinkSet;

interface CompositeStateOwnsSubstateStateVertex :
                                            Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   CompositeStateOwnsSubstateStateVertexLinkSet
     all_composite_state_owns_substate_state_vertex_links();
   boolean exists (in CompositeState parent, in StateVertex substate);
   CompositeState with_substate (in StateVertex substate);
   StateVertexSet with_parent (in CompositeState parent);
   void add (in CompositeState parent, in StateVertex substate)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_parent (in CompositeState parent,
                       in StateVertex substate,
                       in CompositeState new_parent)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_substate (in CompositeState parent,
                         in StateVertex substate,
                         in StateVertex new_substate)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in CompositeState parent, in StateVertex substate)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct TransitionOwnsEffectActionSequenceLink {
   UmlStateMachines::Transition transition;
```

```
    ::UmlCommonBehavior::ActionSequence effect;
 };
 typedef sequence <TransitionOwnsEffectActionSequenceLink>
   TransitionOwnsEffectActionSequenceLinkSet;

interface TransitionOwnsEffectActionSequence : Reflective::RefAssociation {
    readonly attribute UmlStateMachinesPackage enclosing_package_ref;
    TransitionOwnsEffectActionSequenceLinkSet
      all_transition_owns_effect_action_sequence_links();
    boolean exists (in UmlStateMachines::Transition transition,
                    in ::UmlCommonBehavior::ActionSequence effect);
    UmlStateMachines::Transition with_effect (
                           in ::UmlCommonBehavior::ActionSequence effect);
    ::UmlCommonBehavior::ActionSequence with_transition (
                              in UmlStateMachines::Transition transition);
    void add (in UmlStateMachines::Transition transition,
              in ::UmlCommonBehavior::ActionSequence effect)
        raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_transition (in UmlStateMachines::Transition transition,
                            in ::UmlCommonBehavior::ActionSequence effect,
                            in UmlStateMachines::Transition new_transition)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_effect (in UmlStateMachines::Transition transition,
                        in ::UmlCommonBehavior::ActionSequence effect,
                        in ::UmlCommonBehavior::ActionSequence new_effect)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void remove (in UmlStateMachines::Transition transition,
                 in ::UmlCommonBehavior::ActionSequence effect)
        raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
 };

 struct StateOwnsInternalTransitionLink {
    UmlStateMachines::State state;
    Transition internal_transition;
 };
```

```
typedef sequence <StateOwnsInternalTransitionLink>
  StateOwnsInternalTransitionLinkSet;

interface StateOwnsInternalTransition : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateOwnsInternalTransitionLinkSet
     all_state_owns_internal_transition_links();
   boolean exists (in UmlStateMachines::State state,
                   in Transition internal_transition);
   UmlStateMachines::State with_internal_transition (
                                     in Transition internal_transition);
   TransitionSet with_state (in UmlStateMachines::State state);
   void add (in UmlStateMachines::State state,
             in Transition internal_transition)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_state (in UmlStateMachines::State state,
                      in Transition internal_transition,
                      in UmlStateMachines::State new_state)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_internal_transition (in UmlStateMachines::State state,
                                    in Transition internal_transition,
                                  in Transition new_internal_transition)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlStateMachines::State state,
                in Transition internal_transition)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct TransitionIsTriggeredByEventLink {
   UmlStateMachines::Transition transition;
   Event trigger;
};
typedef sequence <TransitionIsTriggeredByEventLink>
  TransitionIsTriggeredByEventLinkSet;
```

```
interface TransitionIsTriggeredByEvent : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   TransitionIsTriggeredByEventLinkSet
     all_transition_is_triggered_by_event_links();
   boolean exists (in UmlStateMachines::Transition transition,
                   in Event trigger);
   UmlStateMachines::TransitionSet with_trigger (in Event trigger);
   Event with_transition (in UmlStateMachines::Transition transition);
  void add (in UmlStateMachines::Transition transition, in Event trigger)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_transition (in UmlStateMachines::Transition transition,
                           in Event trigger,
                         in UmlStateMachines::Transition new_transition)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_trigger (in UmlStateMachines::Transition transition,
                        in Event trigger,
                        in Event new_trigger)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlStateMachines::Transition transition, in Event
     trigger)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct StateMachineOwnsTransitionLink {
   UmlStateMachines::StateMachine state_machine;
   Transition transitions;
};
typedef sequence <StateMachineOwnsTransitionLink>
  StateMachineOwnsTransitionLinkSet;

interface StateMachineOwnsTransition : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   StateMachineOwnsTransitionLinkSet
     all_state_machine_owns_transition_links();
   boolean exists (in UmlStateMachines::StateMachine state_machine,
```

```
                         in Transition transitions);
   UmlStateMachines::StateMachine with_transitions (
                                                    in Transition transitions);
   TransitionSet with_state_machine (
                           in UmlStateMachines::StateMachine state_machine);
   void add (in UmlStateMachines::StateMachine state_machine,
             in Transition transitions)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_state_machine (
                           in UmlStateMachines::StateMachine state_machine,
                            in Transition transitions,
                       in UmlStateMachines::StateMachine new_state_machine)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_transitions (in UmlStateMachines::StateMachine
     state_machine,
                              in Transition transitions,
                              in Transition new_transitions)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlStateMachines::StateMachine state_machine,
                in Transition transitions)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct TransitionHasSourceStateVertexLink {
   Transition outgoing;
   StateVertex source;
};
typedef sequence <TransitionHasSourceStateVertexLink>
  TransitionHasSourceStateVertexLinkSet;

interface TransitionHasSourceStateVertex : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   TransitionHasSourceStateVertexLinkSet
     all_transition_has_source_state_vertex_links();
   boolean exists (in Transition outgoing, in StateVertex source);
```

```
   TransitionSet with_source (in StateVertex source);
   StateVertex with_outgoing (in Transition outgoing);
   void add (in Transition outgoing, in StateVertex source)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_outgoing (in Transition outgoing,
                         in StateVertex source,
                         in Transition new_outgoing)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_source (in Transition outgoing,
                       in StateVertex source,
                       in StateVertex new_source)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Transition outgoing, in StateVertex source)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct TransitionHasTargetStateVertexLink {
   Transition incoming;
   StateVertex target;
};
typedef sequence <TransitionHasTargetStateVertexLink>
  TransitionHasTargetStateVertexLinkSet;

interface TransitionHasTargetStateVertex : Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   TransitionHasTargetStateVertexLinkSet
     all_transition_has_target_state_vertex_links();
   boolean exists (in Transition incoming, in StateVertex target);
   TransitionSet with_target (in StateVertex target);
   StateVertex with_incoming (in Transition incoming);
   void add (in Transition incoming, in StateVertex target)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_incoming (in Transition incoming,
                         in StateVertex target,
                         in Transition new_incoming)
```

```
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_target (in Transition incoming,
                       in StateVertex target,
                       in StateVertex new_target)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in Transition incoming, in StateVertex target)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
};

struct SubmachineStateContainsStateMachineLink {
   StateMachine submachine;
   UmlStateMachines::SubmachineState submachine_state;
};
typedef sequence <SubmachineStateContainsStateMachineLink>
  SubmachineStateContainsStateMachineLinkSet;

interface SubmachineStateContainsStateMachine :
     Reflective::RefAssociation {
   readonly attribute UmlStateMachinesPackage enclosing_package_ref;
   SubmachineStateContainsStateMachineLinkSet
     all_submachine_state_contains_state_machine_links();
   boolean exists (in StateMachine submachine,
                 in UmlStateMachines::SubmachineState submachine_state);
   StateMachine with_submachine_state (
                 in UmlStateMachines::SubmachineState submachine_state);
   UmlStateMachines::SubmachineStateSet with_submachine (
                                            in StateMachine submachine);
   void add (in StateMachine submachine,
             in UmlStateMachines::SubmachineState submachine_state)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_submachine (
                       in StateMachine submachine,
                  in UmlStateMachines::SubmachineState submachine_state,
                           in StateMachine new_submachine)
      raises (Reflective::StructuralError,
```

```
                Reflective::SemanticError,
                Reflective::NotFound);
    void modify_submachine_state (
                     in StateMachine submachine,
                    in UmlStateMachines::SubmachineState submachine_state,
                in UmlStateMachines::SubmachineState new_submachine_state)
       raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
    void remove (in StateMachine submachine,
                  in UmlStateMachines::SubmachineState submachine_state)
       raises (Reflective::StructuralError,
                Reflective::SemanticError,
                Reflective::NotFound);
 };

 struct StateMachineExhibitsBehaviorOfModelElementLink {
    ::UmlCore::ModelElement uml_context;
    StateMachine behavior;
 };
 typedef sequence <StateMachineExhibitsBehaviorOfModelElementLink>
   StateMachineExhibitsBehaviorOfModelElementLinkSet;

 interface StateMachineExhibitsBehaviorOfModelElement :
                                             Reflective::RefAssociation {
    readonly attribute UmlStateMachinesPackage enclosing_package_ref;
    StateMachineExhibitsBehaviorOfModelElementLinkSet
      all_state_machine_exhibits_behavior_of_model_element_links();
    boolean exists (in ::UmlCore::ModelElement uml_context,
                     in StateMachine behavior);
    ::UmlCore::ModelElement with_behavior (in StateMachine behavior);
    StateMachineSet with_uml_context (in ::UmlCore::ModelElement
      uml_context);
    void add (in ::UmlCore::ModelElement uml_context, in StateMachine
      behavior)
       raises (Reflective::StructuralError, Reflective::SemanticError);
    void modify_uml_context (in ::UmlCore::ModelElement uml_context,
                              in StateMachine behavior,
                              in ::UmlCore::ModelElement new_uml_context)
       raises (Reflective::StructuralError,
                Reflective::SemanticError,
```

```
                 Reflective::NotFound);
      void modify_behavior (in ::UmlCore::ModelElement uml_context,
                            in StateMachine behavior,
                            in StateMachine new_behavior)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
      void remove (in ::UmlCore::ModelElement uml_context,
                   in StateMachine behavior)
         raises (Reflective::StructuralError,
                 Reflective::SemanticError,
                 Reflective::NotFound);
   };

   interface UmlStateMachinesPackageFactory  {
      UmlStateMachinesPackage create_uml_state_machines_package ()
         raises (Reflective::SemanticError);
   };

   interface UmlStateMachinesPackage : Reflective::RefPackage {
      readonly attribute EventClass event_class_ref;
      readonly attribute CallEventClass call_event_class_ref;
      readonly attribute ChangeEventClass change_event_class_ref;
      readonly attribute SignalEventClass signal_event_class_ref;
      readonly attribute TimeEventClass time_event_class_ref;
      readonly attribute StateVertexClass state_vertex_class_ref;
      readonly attribute GuardClass guard_class_ref;
      readonly attribute TransitionClass transition_class_ref;
      readonly attribute PseudostateClass pseudostate_class_ref;
      readonly attribute StateClass state_class_ref;
      readonly attribute CompositeStateClass composite_state_class_ref;
      readonly attribute PartitionClass partition_class_ref;
      readonly attribute ClassifierInStateClass
        classifier_in_state_class_ref;
      readonly attribute StateMachineClass state_machine_class_ref;
      readonly attribute ActivityModelClass activity_model_class_ref;
      readonly attribute SimpleStateClass simple_state_class_ref;
      readonly attribute ActivityStateClass activity_state_class_ref;
      readonly attribute ObjectFlowStateClass object_flow_state_class_ref;
      readonly attribute ActionStateClass action_state_class_ref;
      readonly attribute SubmachineStateClass submachine_state_class_ref;
```

```
    readonly attribute StateOwnsEntryActionSequence
      state_owns_entry_action_sequence_ref;
    readonly attribute StateOwnsExitActionSequence
      state_owns_exit_action_sequence_ref;
    readonly attribute TransitionOwnsGuard transition_owns_guard_ref;
    readonly attribute SignalEventIsOccurrenceOfSignal
      signal_event_is_occurrence_of_signal_ref;
    readonly attribute ActivityModelOwnsPartition
      activity_model_owns_partition_ref;
    readonly attribute PartitionHasContextOfModelElement
      partition_has_context_of_model_element_ref;
    readonly attribute ClassifierInStateIsInStateState
      classifier_in_state_is_in_state_state_ref;
    readonly attribute ObjectFlowStateRepresentsClassifierInState
      object_flow_state_represents_classifier_in_state_ref;
    readonly attribute ClassifierInStateIsOfTypeClassifier
      classifier_in_state_is_of_type_classifier_ref;
    readonly attribute StateMachineOwnsTopState
      state_machine_owns_top_state_ref;
    readonly attribute StateDefersEvent state_defers_event_ref;
    readonly attribute CallEventIsOccurrenceOfOperation
      call_event_is_occurrence_of_operation_ref;
    readonly attribute CompositeStateOwnsSubstateStateVertex
      composite_state_owns_substate_state_vertex_ref;
    readonly attribute TransitionOwnsEffectActionSequence
      transition_owns_effect_action_sequence_ref;
    readonly attribute StateOwnsInternalTransition
      state_owns_internal_transition_ref;
    readonly attribute TransitionIsTriggeredByEvent
      transition_is_triggered_by_event_ref;
    readonly attribute StateMachineOwnsTransition
      state_machine_owns_transition_ref;
    readonly attribute TransitionHasSourceStateVertex
      transition_has_source_state_vertex_ref;
    readonly attribute TransitionHasTargetStateVertex
      transition_has_target_state_vertex_ref;
    readonly attribute SubmachineStateContainsStateMachine
      submachine_state_contains_state_machine_ref;
    readonly attribute StateMachineExhibitsBehaviorOfModelElement
      state_machine_exhibits_behavior_of_model_element_ref;
```

```
    };
};
```

5.4.7 UMLUseCases

```
#include "UmlCommonBehavior.idl"

module UmlUseCases {
   interface UmlUseCasesPackage;
   interface ExtensionPoint;
   interface ExtensionPointClass;
   typedef sequence<ExtensionPoint> ExtensionPointUList;
   typedef sequence<ExtensionPoint> ExtensionPointSet;
   interface UseCase;
   interface UseCaseClass;
   typedef sequence<UseCase> UseCaseUList;
   interface Actor;
   interface ActorClass;
   typedef sequence<Actor> ActorUList;
   interface UseCaseInstance;
   interface UseCaseInstanceClass;
   typedef sequence<UseCaseInstance> UseCaseInstanceUList;

   interface ActorClass : ::UmlCore::ClassifierClass {
      readonly attribute ActorUList all_of_kind_actor;
      readonly attribute ActorUList all_of_type_actor;
      Actor create_actor (in ::UmlCore::Name name,
                          in boolean is_root,
                          in boolean is_leaf,
                          in boolean is_abstract)
         raises (Reflective::SemanticError);
   };

   interface Actor : ActorClass, ::UmlCore::Classifier { };

   interface UseCaseClass : ::UmlCore::ClassifierClass {
      readonly attribute UseCaseUList all_of_kind_use_case;
      readonly attribute UseCaseUList all_of_type_use_case;
      UseCase create_use_case (in ::UmlCore::Name name,
                               in boolean is_root,
                               in boolean is_leaf,
```

```
                            in boolean is_abstract)
      raises (Reflective::SemanticError);
};

interface UseCase : UseCaseClass, ::UmlCore::Classifier {
   UmlUseCases::ExtensionPointSet extension_point ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void add_extension_point (in UmlUseCases::ExtensionPointSet new_value)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void remove_extension_point ()
      raises (Reflective::SemanticError);
};

interface UseCaseInstanceClass : ::UmlCommonBehavior::UmlInstanceClass {
   readonly attribute UseCaseInstanceUList all_of_kind_use_case_instance;
   readonly attribute UseCaseInstanceUList all_of_type_use_case_instance;
   UseCaseInstance create_use_case_instance (in ::UmlCore::Name name)
      raises (Reflective::SemanticError);
};

interface UseCaseInstance : UseCaseInstanceClass,
                            ::UmlCommonBehavior::UmlInstance { };

interface ExtensionPointClass : ::UmlCore::ModelElementClass {
   readonly attribute ExtensionPointUList all_of_kind_extension_point;
   readonly attribute ExtensionPointUList all_of_type_extension_point;
   ExtensionPoint create_extension_point (in ::UmlCore::Name name,
                                          in ::UmlCore::Expression body)
      raises (Reflective::SemanticError);
};

interface ExtensionPoint : ExtensionPointClass, ::UmlCore::ModelElement {
   ::UmlCore::Expression body ()
      raises (Reflective::SemanticError);
   void set_body (in ::UmlCore::Expression new_value)
      raises (Reflective::SemanticError);
   UmlUseCases::UseCase use_case ()
      raises (Reflective::NotSet, Reflective::SemanticError);
   void set_use_case (in UmlUseCases::UseCase new_value)
      raises (Reflective::SemanticError);
   void unset_use_case ()
```

```
      raises (Reflective::SemanticError);
};

struct UseCaseOwnsExtensionPointLink {
   UmlUseCases::UseCase use_case;
   UmlUseCases::ExtensionPoint extension_point;
};
typedef sequence <UseCaseOwnsExtensionPointLink>
  UseCaseOwnsExtensionPointLinkSet;

interface UseCaseOwnsExtensionPoint : Reflective::RefAssociation {
   readonly attribute UmlUseCasesPackage enclosing_package_ref;
   UseCaseOwnsExtensionPointLinkSet
        all_use_case_owns_extension_point_links();
   boolean exists (in UmlUseCases::UseCase use_case,
                   in UmlUseCases::ExtensionPoint extension_point);
   UmlUseCases::UseCase with_extension_point (
                          in UmlUseCases::ExtensionPoint extension_point);
   UmlUseCases::ExtensionPointSet with_use_case (
                              in UmlUseCases::UseCase use_case);
   void add (in UmlUseCases::UseCase use_case,
             in UmlUseCases::ExtensionPoint extension_point)
      raises (Reflective::StructuralError, Reflective::SemanticError);
   void modify_use_case (in UmlUseCases::UseCase use_case,
                         in UmlUseCases::ExtensionPoint extension_point,
                         in UmlUseCases::UseCase new_use_case)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void modify_extension_point (
                          in UmlUseCases::UseCase use_case,
                          in UmlUseCases::ExtensionPoint extension_point,
                      in UmlUseCases::ExtensionPoint new_extension_point)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
   void remove (in UmlUseCases::UseCase use_case,
                in UmlUseCases::ExtensionPoint extension_point)
      raises (Reflective::StructuralError,
              Reflective::SemanticError,
              Reflective::NotFound);
```

```
    };

    interface UmlUseCasesPackageFactory  {
       UmlUseCasesPackage create_uml_use_cases_package ()
          raises (Reflective::SemanticError);
    };

    interface UmlUseCasesPackage : Reflective::RefPackage {
       readonly attribute ActorClass actor_class_ref;
       readonly attribute UseCaseClass use_case_class_ref;
       readonly attribute UseCaseInstanceClass use_case_instance_class_ref;
       readonly attribute ExtensionPointClass extension_point_class_ref;

       readonly attribute UseCaseOwnsExtensionPoint
         use_case_owns_extension_point_ref;
    };
};
```

UML Standard Elements A

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 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.

Name	Applies to	Description
«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.

Name	Applies to	Description
«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 supplier so that the client can see into the supplier.
«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.

Name	Applies to	Description
«implementation Class»	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.

Name	Applies to	Description
«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 that specifies private inheritance. It hides the inherited features of a class and 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 marked explicitly.

Name	Applies to	Description
«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 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.

Name	Applies to	Description
«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 no 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).

Name	Applies to	Description
	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 additional subtypes are not 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.

Name	Applies to	Description
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.

Object Constraint Language B

B.4 Overview

This sppendix introduces and defines the Object Constraint Language (OCL), a formal language to express side-effect-free constraints. Users of the Unified Modeling Language and other languages can use OCL to specify constraints and other expressions attached to their models.

OCL was used in the UML Semantics chapter to specify the well-formedness rules of the UML metamodel. Each well-formedness rule in the static semantics chapters in the UML Semantics section contains an OCL expression, which is an invariant for the involved class. The grammar for OCL is specified at the end of this chapter. A parser generated from this grammar has correctly parsed all the constraints in the UML Semantics section, a process which improved the correctness of the specifications for OCL and UML.

B.1.1 Why OCL?

In object-oriented modeling a graphical model, like a class model, is not enough for a precise and unambiguous specification. There is a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a string mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method.

OCL is a pure expression language; therefore, an OCL expression is guaranteed to be without side effect. It cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition). All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value.

OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable.

OCL is a typed language, so each OCL expression has a type. In a correct OCL expression, all types used must be type conformant. For example, you cannot compare an Integer with a String. Types within OCL can be any kind of Classifier within UML.

As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL. Each OCL expression is conceptually atomic. The state of the objects in the system cannot change during evaluation.

B.1.2 Where to Use OCL

OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- As a navigation language
- To specify constraints on operations:

  ```
  operation = expression
  ```

  Where operation is the name of the operation and expression the constraint. Because operations may have parameters, the constraint may also have one or more parameters, as in one of the following:

  ```
  operation(a, b) = expression
  operation(a : Type1, b : Type2) = expression
  ```

  The parameters of the operation, in this example *a* and *b*, can be used in the expression at the right-hand side of the equals sign. Operations can also be described by a recursive expression. It is the modeler’s task to make sure that the recursion is well defined. An operation constraint can also be read as a definition of the operation, where the right-hand side of the equals sign determines the value that the operation will return.

Within the UML Semantics chapter, OCL is used in the well-formedness rules as invariants on the meta-classes in the abstract syntax. In several places, it is also used to define ‘additional’ operations which are used in the well-formedness rules.

B.5 Introduction

B.2.3 Legend

Text written in the courier typeface as shown below is an OCL expression.

```
'This is an OCL expression'
```

The underlined word before an OCL expression determines the context for the expression.

```
TypeName
    'this is an OCL expression in the context of TypeName'
```

Keywords of OCL are written in boldface within the OCL expression in this document. The boldface has no formal meaning, but is used to make the expressions more readable in this document. OCL expressions are written using only ASCII characters.

Words in *Italics* within the main text of the paragraphs refer to parts of OCL expressions.

B.2.4 Example Class Diagram

The diagram below is used in the examples in this document.

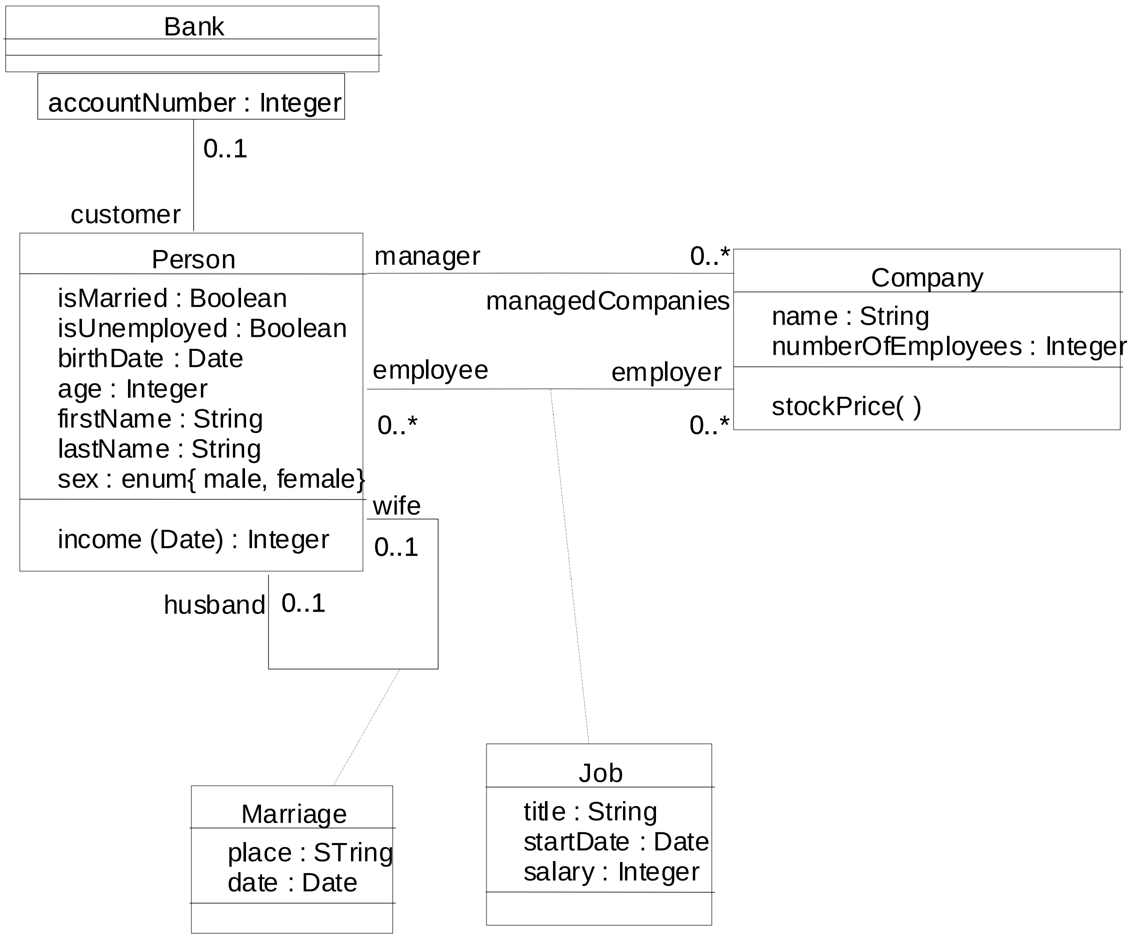


Figure 2-1 Class Diagram Example

B.6 Connection with the UML Metamodel

B.3.1 Self

Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the name *self* is used to refer to the contextual instance.

B.3.2 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped with «invariant». When the Invariant is associated with a Classifier, this is called the type in this document. The expression then is an invariant of the type and must be true for all instances of that type at any time. If the context is Company, then *self* refers to an instance of Company.

In the expression:

```
self.numberOfEmployees
```

self is an instance of type Company. We can see the *self* as the object from where we start the expression.

In this document, the type of the contextual instance of an OCL expression, which is part of an Invariant, is written with the name of the type underlined as follows:

```
Company
    self.numberOfEmployees
```

In most cases, *self* can be left out because the context is clear, as in the above examples.

As an alternative for self, a different name can be defined playing the part of self:

```
c : Company
    c.numberOfEmployees
```

This is identical to the previous example using self.

B.3.3 Pre and Postconditions

The OCL expression can be part of a Precondition or Postcondition, which are Constraints stereotyped with respectively «precondition» and «postcondition», the Precondition or Postcondition on Operation or Method. In this case, the expression is a pre- or postcondition on the Operation or Method. The contextual instance *self* then is of the type which owns the operation as a feature. The notation used in this document is to underline the type and operation declaration, and to put labels 'pre:' and 'post:' before Preconditions and Postconditions

```
Typename::operationName(parameter1 : Type1, ... ): ReturnType
    pre :  parameter1 > …
    post:  result = ...
```

The name *self* can be used in the expression referring to the object on which the operation was called. The name *result* is the name of the returned object, if there is any. The names of the parameters (*parameter1,*) can also be used in the OCL expression. In the example diagram, we can write:

```
Person::income(d : Date) : Integer
    post:  result = ...some function of self and parameter1 ...
```

B.3.4 Guards

The OCL expression can be part of a Guard. In this case, *self* refers to the enclosing Classifier. No examples of guards are given in this document.

B.3.5 General Expressions

Any OCL expression can be used as the value for an attribute of the UML class Expression or one of its subtypes. In this case, the semantics section describes the meaning of the expression.

B.7 Basic Values and Types

In OCL, a number of basic types are predefined and available to the modeler at all time. These predefined value types are independent of any object model and part of the definition of OCL.

The most basic value in OCL is a value of one of the basic types. Some basic types used in the examples in this document, with corresponding examples of their values, are shown in Table 2-1.

Table 2-1 Basic Values and Types

type	values
Boolean	true, false
Integer	1, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

OCL defines a number of operations on the predefined types. Table 2-2 gives some examples of the operations on the predefined types. See “Predefined OCL Types” on page B-23 for a complete list of all operations.

Table 2-2 Operations on Predefined Types

type	operations
Integer	*, +, -, /, abs
Real	*, +, -, /, floor
Boolean	and, or, xor, not, implies, if-then-else
String	toUpper, concat

The complete list of operations provided for each type is described at the end of this chapter. Collection, Set, Bag and Sequence are basic types as well. Their specifics will be described in the upcoming sections.

B.4.1 Types from the UML Model

Each OCL expression is written in the context of a UML model, a number of types/classes, their features and associations, and their generalizations. All types/classes from the UML model are types in OCL that is attached to the model.

B.4.2 Enumeration Types

As shown in the example diagram, new enumeration types can be defined in a model by using:

```
enum{ value1, value2, value3 }
```

The values of the enumeration (*value1*, ...) can be used within expressions.

As there might be a name conflict with attribute names being equal to enumeration values, the usage of an enumeration value is expressed syntactically with an additional # symbol in front of the value:

```
#value1
```

The type of an enumeration attribute is Enumeration, with restrictions on the values for the attribute.

B.4.3 Type Conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines conformance of the different types to each other. You cannot, for example, compare an Integer with a Boolean or a String.

An OCL expression in which all the types conform is a valid expression. An OCL expression in which the types don't conform is an invalid expression. It contains a type *conformance error*. A type *type1* conforms to a type *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected. The type conformance rules for types in the class diagrams are simple.

- Each type conforms to its supertype.
- Type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

The effect of this is that a type conforms to its supertype, and all the supertypes above. The type conformance rules for the value types are listed in Table 2-3.

Table 2-3 Type Conformance Rules for Value Types

Type	**Conforms to/Is subtype of**
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See "Collection Type Hierarchy and Type Conformance Rules" on page B-17 for the complete conformance rules for collections.

Table 2-4 provides examples of valid and invalid expressions.

Table 2-4 Valid and Invalid Expression Examples

OCL expression	**valid?**	**error**
1 + 2 * 34	yes	
1 + 'motorcycle'	no	type Integer does not conform to type String
23 * false	no	type Integer does not conform to Boolean
12 + 13.5	yes	

B.4.4 Re-typing or Casing

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation *oclAsType(OclType)*. This operation results in the same object, but the known type is the argument *OclType*. When there is an object *object* of type *Type1* and *Type2* is another type, it is allowed to write:

```
object.oclAsType(Type2)    --- evaluates to object with type Type2
```

An object can only be re-typed to one of its subtype; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not equal to the type to which it is re-typed, the expression is undefined (see "Undefined Values" on page B-9).

B.4.5 Precedence Rules

The precedence order for the operations in OCL is:

- dot and arrow operations have highest precedence
- unary 'not' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- 'and', 'or' and 'xor'
- 'implies'
- 'if-then-else-endif'
- '<', '>', '<=', '>=' and '='

Parenthesis '(' and ')' can be used to change precedence.

B.4.6 Comment

Comments in OCL are written after two dashes. Everything after the two dashes up to and including the end of line is comment. For example:

```
-- this is a comment
```

B.4.7 Undefined Values

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

There are two exceptions to this for the boolean operators:

- True OR-ed with anything is True
- False AND-ed with anything is False

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other sub-expression is known.

B.8 Objects and Properties

OCL expressions can refer to types, classes, interfaces, associations (acting as types) and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used. In a class model, an operation or method is defined to be side-effect-free if the isQuery attribute of the operations is true. For the purpose of this document, we will refer to attributes, association-ends, and side-effect-free methods and operations as being properties. A property is one of:

- an Attribute
- an AssociationEnd
- an Operation with *isQuery* being true
- a Method with *isQuery* being true

B.5.1 Properties

The value of a property on an object that is defined in a class diagram is specified by a dot followed by the name of the property.

<u>AType</u>

```
self.property
```

If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self*.

B.5.2 Properties: Attributes

For example, the age of a Person is written as:

```
Person
    self.age
```

The value of this expression is the value of the *age* attribute on the Person *self*. The type of this expression is the type of the attribute *age*, which is the basic type Integer.

With of attributes, and the operations defined on the basic value types, we can express calculations etc. over the class model. For example, a business rule might be "the age of a Person is always greater or equal to zero." This can be stated as the invariant:

```
Person
    self.age >= 0
```

B.5.3 Properties: Operations

Operations may have parameters. For example, as shown earlier, a Person object has an income expressed as a function of the date. This operation would be accessed as follows, for a Person *aPerson* and a date *aDate*:

```
    aPerson.income(aDate)
```

The operation itself could be defined by a postcondition constraint. This is a constraint that is stereotyped as «postcondition». The object that is returned by the operation can be referred to by *result*. It takes the following form:

```
Person::income (d: Date) : Integer
    post: result = - - some function of d and other properties of
person
```

The right-hand-side of this definition may refer to the operation being defined (i.e., the definition may be recursive) as long as the recursion is well defined. The type of *result* is the return type of the operation, which is Integer in the above example.

To refer to an operation or a method that doesn't take a parameter, parentheses with an empty argument list are used:

```
Company
    self.stockPrice()
```

B.5.4 Properties: Association Ends and Navigation

Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end:

```
    object.rolename
```

The value of this expression is the set of objects on the other side of the *rolename* association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. In the example class diagram, when we start in the context of a Company (i.e., *self* is an instance of Company), we can write:

```
Company
    self.manager   -- is of type      Person
    self.employee  -- is of type      Set(Person)
```

The evaluation of the first expression will result in an object of type Person, because the multiplicity of the association is one. The evaluation of the second expression will result in a Set of Persons. By default, navigation will result in a Set. When the association on the Class Diagram is adorned with {ordered}, the navigation results in a Sequence.

Collections, like Sets, Bags, and Sequences are predefined types in OCL. They have a large number of predefined operations on them. A property of the collection itself is accessed by using an arrow '->' followed by the name of the property. The following example is in the context of a person:

```
Person
    self.employer->size
```

This applies the *size* property on the Set *self.employer*, which results in the number of employers of the Person *self*.

```
Person
    self.employer->isEmpty
```

This applies the *isEmpty* property on the Set *self.employer.* This evaluates to true if the set of employers is empty and false otherwise.

Missing Rolenames

Whenever a rolename is missing at one of the ends of an association, the name of the type at the association end, starting with a lowercase character, is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

Navigation over Associations with Multiplicity Zero or One

Because the multiplicity of the role manager is one, *self.manager* is an object of type Person. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set. This is shown in the following example:

```
Company
    self.manager->size -- 'self.manager' is used as Set, because the
                       -- arrow
```

```
                    -- is used to access the 'size' property on
                       Set
                    -- This expresin result in 1

self.manager->foo   -- self.manager' is used as Set, because the
                    --  arrow is used to access the 'foo' property
                    --  on Set.This expression is incorrect,
                    --  since 'foo' is not a defined property of
                    --  Set.

self.manager.age    -- 'self.manager' is used as Person, because
                    -- the dot is used to access the 'age'
                    -- property of Person
```

In the case of an optional (0..1 multiplicity) association, this is especially useful to check whether there is an object or not when navigating the association. In the example we can write:

<u>Company</u>

```
self.wife->notEmpty implies self.wife.sex = female
```

Combining Properties

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to a specific object of a specific type. Upon this result, one can always apply another property. Therefore, each OCL expression can be read and evaluated left-to-right.

Following are some invariants that use combined properties on the example class diagram:

[1] Married people are of age >= 18

```
self.wife->notEmpty implies self.wife.age >= 18 and
self.husband->notEmpty implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
self.employee->size <= 50
```

[3] A marriage is between a female (wife) and male (husband)

```
self.wife.sex = #female and
self.husband.sex = #male
```

[4] A person cannot both have a wife and a husband

```
not ((self.wife->size = 1) and (self.husband->size = 1))
```

B.5.5 Navigation to Association Types

To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

```
Person
    self.job
```

This evaluates to a Set of all the jobs a person has with the companies that are his/her employer. In the case of an association class, there is no explicit rolename in the class diagram. The name *job* used in this navigation is the name of the association class starting with a lowercase character, similar to the way described in the section "Missing Rolenames" above.

B.5.6 Navigation from Association Classes

We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends.

```
Job
    self.employer
    self.employee
```

Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->).

B.5.7 Navigation through Qualified Associations

Qualified associations use one or more qualifier attributes to select the objects at the other end of the association. To navigate them, we can add the values for the qualifiers to the navigation. This is done using square brackets, following the role-name. It is permissible to leave out the qualifier values, in which case the result will be all objects at the other end of the association.

```
Bank
    self.customer          --  results in a Set(Person) containing
                           --  all customers of the Bank
    self.customer[8764423] --  results in one Person, having account
                           --  number 8764423
```

If there is more than one qualifier attribute, the values are separated by commas. It is not permissible to partially specify the qualifier attribute values.

B.5.8 Using Pathnames for Packages and Properties

Within UML, different types are organized in packages. OCL provides a way of explicitly referring to types in other packages by using a package-pathname prefix. The syntax is a package name, followed by a double colon:

```
Packagename::Typename
```

This usage of pathnames is transitive and can also be used for packages within packages:

```
Packagename1::Packagename2::Typename
```

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the same path syntax. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

```
B
    self.A::p1    -- accesses the p1 property defined in A
    self.B::p1    -- accesses the p1 property defined in B
```

Figure 2-2 shows an example where such a pathname is needed.

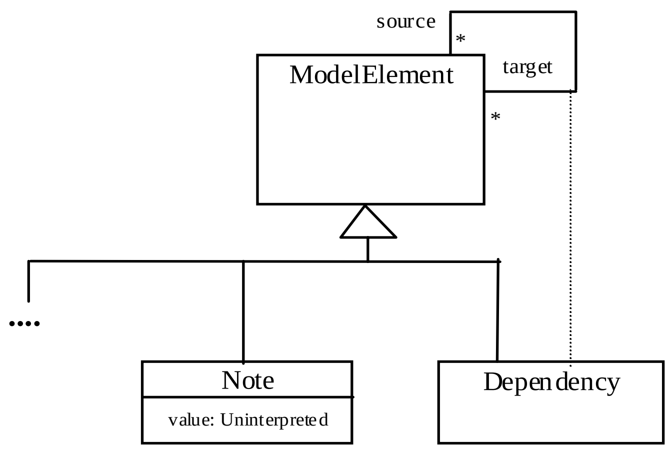


Figure 2-2 Pathname Example

In this model fragment there is an ambiguity with the OCL expression on Dependency:

```
Dependency
  self.source
```

This can either mean normal association navigation, which is inherited from ModelElement, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using the pathname we can distinguish between them with:

```
Dependency
    self.Dependency::source
    self.ModelElement::source
```

B.5.9 Predefined Features on All Objects

There are several features that apply to all objects, and are predefined in OCL. These are:

```
oclType                  : OclType
oclIsTypeOf(t : OclType) : boolean
oclIsKindOf(t : OclType) : boolean
```

The feature *oclType* results in the type of an object. For example, the expression

```
Person
    self.oclType
```

results in Person. The type of this is OclType, a predefined type within the OCL language.

Note – not Person, which is the type of self.

The operation is*TypeOf* results in true if the *type* of self and *t* are the same. For example:

```
Person
    self.oclIsTypeOf( Person )      -- is true
    self.oclIsTypeOf( Company)      -- is false
```

The above feature deals with the direct type of an object. The *oclIsKindOf* feature determines whether *t* is either the direct type or one of the supertypes of an object.

B.5.10 Features on Types Themselves

All properties discussed until now in OCL are properties on instances of classes. The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the *class*-scoped features defined in the class model. Furthermore, several features are predefined on each type.

The most important predefined feature on each type is *allInstances*, which results in the Set of all instances of the type. If we want to make sure that all instances of Person have unique names, we can write:

```
Person.allInstances->forAll(p1, p2 | p1 <> p2 implies p1.name <>
p2.name)
```

The *Person.allInstances* is the set of all persons and is of type Set(Person).

B.5.11 Collections

Navigation will most often result in a collection; therefore, the collection types play an important role in OCL expressions.

The type Collection is predefined in OCL. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; *isQuery* is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one.

Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates (i.e., the same element may be in a bag twice or more). A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. Sets, Sequences, and Bags can be specified by a literal in OCL. Curly brackets surround the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
```

A Sequence:

```
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
```

A bag:

```
Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a Sequence of consecutive Integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer, *Int-expr1* and *Int-expr2*, separated by '..'. This denotes all the Integers between the values of *Int-expr1* and *Int-expr2*, including the values of *Int-expr1* and *Int-expr2* themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

The complete list of Collection operations is described at the end of this chapter.

Collections can be specified by a literal, as described above. The only other way to get a collection is by navigation. To be more precise, the only way to get a Set, Sequence, or Bag is:

1. a literal, this will result in a Set, Sequence, or Bag:

```
Set      {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Sequence {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
Bag      {1, 2, 3, 2, 1}
```

2. a navigation starting from a single object can result in a collection:

```
Company
```

```
self.employee
```

3. operations on collections may result in new collections:

```
collection1->union(collection2)
```

B.5.12 Collections of Collections

Within OCL, all Collections of Collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

B.5.13 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in “Type Conformance” on page B-7, the following rules hold for all types, including the collection types:

- Every type Collection (X) is a subtype of OclAny. The types Set (X), Bag (X) and Sequence (X) are all subtypes of Collection (X).

Type conformance rules are as follows for the collection types:

- *Type1* conforms to *Type2* when they are identical (standard rule for all types).
- *Type1* conforms to *Type2* when it is a subtype of Type2 (standard rule for all types).
- *Collection(Type1)* conforms to *Collection(Type2)*, when *Type1* conforms to *Type2*.
- Type conformance is transitive: if *Type1* conforms to *Type2*, and *Type2* conforms to *Type3*, then *Type1* conforms to *Type3* (standard rule for all types).

For example, if *Bicycle* and *Car* are two separate subtypes of *Transport:*

```
Set(Bicycle)  conforms to  Set(Transport)
Set(Bicycle)  conforms to  Collection(Bicycle)
Set(Bicycle)  conforms to  Collection(Transport)
```

Note that Set(Bicycle) does not conform to Bag(Bicycle), nor the other way around. They are both subtypes of Collection(Bicycle) at the same level in the hierarchy.

B.5.14 Previous Values in Postconditions

As stated in “Pre and Postconditions” on page B-5, OCL can be used to specify pre- and post-conditions on Operations and Methods in UML. In a postcondition, the expression can refer to two sets of values for each property of an object:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property-name with the commercial at sign '@' followed by the keyword 'pre':

```
Person::birthdayHappens()
   post: age = age@pre + 1
```

The property *age* refers to the property of the instance of Person on which executes the operation. The property *age@pre* refers to the value of the property *age* of the Person that executes the operation, at the start of the operation.

If the property has parameters, the '@pre' is postfixed to the propertyname, before the parameters.

```
Company::hireEmployee(p : Person)
   post: employees = employees@pre->including(p) and
         stockprice() = stockprice@pre() + 10
```

The above operation can also be specified by a post and pre condition together:

```
Company::hireEmployee(p : Person)
   pre : not employee->includes(p)
   post: employees->includes(p) and
         stockprice() = stockprice@pre() + 10
```

When the pre-value of a property is takes and this evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c          -- takes the old value of property b of a, say x
                   -- and then the new value of c of x.
a.b@pre.c@pre      -- takes the old value of property b of a, say x
                   -- and then the old value of c of x.
```

The '@pre' postfix is allowed only in OCL expressions that are part of a Postcondition. Asking for a current property of an object that has been destroyed during execution of the operation results in Undefined. Also, referring to the previous value of an object that has been created during execution of the operation results in Undefined.

B.9 Collection Operations

OCL defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones. The different constructs are described in the following sections.

B.6.1 Select and Reject Operations

Sometimes an expression using operations and navigations delivers a collection, while we are interested only in a special subset of the collection. OCL has special constructs to specify a selection from a specific collection. These are the *select* and *reject* operations. The select specifies a subset of a collection. A select is an operation on a collection and is specified using the arrow-syntax:

```
collection->select( ... )
```

The parameter of select has a special syntax that enables one to specify which elements of the collection we want to select. There are three different forms, of which the simplest one is:

```
collection->select( boolean-expression )
```

This results in a collection that contains all the elements from *collection* for which the *boolean-expression* evaluates to true. To find the result of this expression, for each element in *collection* the expression *boolean-expression* is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not. As an example, the next OCL expression specifies all the employees older than 50 years:

```
Company
    self.employee->select(age > 50)
```

The *self.employee* is of type Set(Person). The *select* takes each person from *self.employee* and evaluates *age > 50* for this person. If this results in *true*, then the person is in the result Set.

As shown in the previous example, the context for the expression in the select argument is the element of the collection on which the select is invoked. Thus the *age* property is taken in the context of a person.

In the above example, it is impossible to refer explicitly to the persons themselves; you can only refer to properties of them. To enable to refer to the persons themselves, there is a more general syntax for the select expression:

```
Collection->select( v | boolean-expression-with-v )
```

The variable *v* is called the iterator. When the select is evaluated, *v* iterates over the *collection* and the *boolean-expression-with-v* is evaluated for each *v*. The *v* is a reference to the object from the collection and can be used to refer to the objects themselves from the *collection*. The two examples below are identical:

```
Company
    self.employee->select(age > 50)
Company
    self.employee->select(p | p.age > 50)
```

The result of the complete select is the collection of persons *p* for which the *p.age > 50* evaluates to True. This amounts to a subset of *self.employee*.

As a final extension to the select syntax, the expected type of the variable v can be given. The select now is written as:

```
Collection->select( v : Type | boolean-expression-with-v )
```

The meaning of this is that the objects in *collection* must be of type *Type*. The next example is identical to the previous examples:

Company

```
self.employee.select(p : Person | p.age > 50)
```

The compete select syntax now looks like one of:

```
collection->select( v : Type | boolean-expression-with-v )
collection->select( v | boolean-expression-with-v )
collection->select( boolean-expression )
```

The *Reject* operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to **False**. The reject syntax is identical to the select syntax:

```
Collection->reject( v : Type | boolean-expression-with-v )
Collection->reject( v | boolean-expression-with-v )
Collection->reject( boolean-expression )
```

As an example, specify all the employees who are **not** married:

Company

```
self.employee->reject( isMarried )
```

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

```
Collection->reject( v : Type | boolean-expression-with-v )
collection->select( v : Type  | not (boolean-expression-with-v) )
```

B.6.2 Collect Operation

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a collect operation. The collect operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

The value of the reject operation is the collection of the results of all the evaluations of *expression-with-v*.

An example: specify the collection of *birthDates* for all employees in the context of a company. This can be written as one of:

Company

```
self.employee->collect( birthDate )
self.employee->collect( person | person.birthDate )
self.employee->collect( person : Person | person.birthDate )
```

An important issue here is that the resulting collection is not a Set, but a Bag. When more than one employee has the same value for *birthDate*, this value will be an element of the resulting Bag more than once. The Bag resulting from the *collect* operation always has the same size as the original collection.

It is possible to make a Set from the Bag, by using the asSet property on the Bag. The following expression results in the Set of different *birthDates* from all employees of a Company:

```
Company
    self.employee->collect( birthDate )->asSet
```

Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable. Instead of

```
self.employee->collect(birthdate)
```

we can also write:

```
self.employee.birthdate
```

In general, when we apply a property to a Collection of Objects, then it will automatically be interpreted as a **collect** over the members of the Collection with the specified property.

For any *propertyname* that is defined as a property on the objects in a collection, the following two expressions are identical:

```
collection.propertyname
collection->collect(propertyname)
```

and so are these if the property is parameterized:

```
collection.propertyname(par1, par2, …)
collection->collect(propertyname(par1, par2, …)
```

B.6.3 ForAll Operation

Many times a constraint is needed on all elements of a collection. The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll( v : Type | boolean-expression-with-v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

This forAll expression results in a Boolean. The result is true if the *boolean-expression-with-v* is true for all elements of *collection*. If the *boolean-expression-with-v* is false for one or more *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
Company
    self.employee->forAll( forename = 'Jack' )
```

```
self.employee->forAll( p | p.forename = 'Jack' )
self.employee->forAll( Person p | p.forename = 'Jack' )
```

These expressions evaluate to true if the forename feature of each employee is equal to 'Jack.'

The forAll operation has an extended variant in which more then one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a forAll on the Cartesian product of the collection with itself.

```
Company
    self.employee->forAll( e1, e2 |
                    e1 <> e2 implies e1.forename <> e2.forename)
    self.employee->forAll(Person e1, e2 |
                    e1 <> e2 implies e1.forename <> e2.forename)
```

This expression evaluates to true if the forenames of all employees are different. It is semantically equivalent to:

```
Company
  self.employee->forAll(e1 | self.employee->forAll (e2 |
                    e1 <> e2 implies e1.forename <> e2.forename)))
```

B.6.4 Exists Operation

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The exists operation in OCL allows you to specify a boolean expression which must hold for at least one object in a collection:

```
collection->exists( v : Type | boolean-expression-with-v )
collection->exists( v | boolean-expression-with-v )
collection->exists( boolean-expression )
```

This forAll operation results in a Boolean. The result is true if the *boolean-expression-with-v* is true for at least one element of *collection*. If the *boolean-expression-with-v* is false for all *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
Company
    self.employee->exists( forename = 'Jack' )
    self.employee->exists( p | p.forename = 'Jack' )
    self.employee->exists( p : Person | p.forename = 'Jack' )
```

These expressions evaluate to true if the forename feature of at least one employee is equal to 'Jack.'

B.6.5 Iterate Operation

The *iterate* operation is slightly more complicated, but is very generic. The operations *reject, select, forAll, exists, collect, elect* can all be described in terms of *iterate*.

An accumulation builds one value by iterating over a collection.

```
    collection->iterate( elem : Type; acc : Type = <expression> |
                             expression-with-elem-and-acc )
```

The variable *elem* is the iterator, as in the definition of *select*, *forAll*, etc. The variable *acc* is the accumulator. The accumulator gets an initial value <*expression*>.

When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with-elem-and-acc* is evaluated for each *elem*. After each evaluation of *expression-with-elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection. The collect operation described in terms of iterate will look like:

```
    collection->collect(x : T | x.property)
-- is identical to:
    collection->iterate(x : T; acc : T2 = Bag{} |
                                    acc->including(x.property))
```

Or written in Java-like pseudocode the result of the iterate can be calculated as:

```
iterate(elem : T; acc : T2 = value)
{
   acc = value;
   for(Enumeration e = collection.elements() ; e.hasMoreElements();
){
       elem = e.nextElement();
       acc  = <expression-with-elem-and-acc>
   }
}
```

B.10 Predefined OCL Types

This section contains all standard types defined within OCL, including all the features defined on those types. Its signature and a description of its semantics define each feature. Within the description, the name ‘result’ is used to refer to the value that results from evaluating the feature. In several places, post conditions are used to describe properties of the result. When there is more than one postcondition, all postconditions must be true.

B.7.1 Basic Types

The basic types used are Integer, Real, String, and Boolean. They are supplemented with OclExpression, OclType, and OclAny.

OclType

All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler access to the meta-level of the model. This can be useful for advanced modelers.

Features of OclType, the instance of OclType is called t*ype*.

`type.name : String`

The name of *type*.

`type.attributes : Set(String)`

The set of names of the attributes of *type*, as they are defined in the model.

`type.associationEnds : Set(String)`

The set of names of the navigable associationEnds of *type*, as they are defined in the model.

`type.operations : Set(String)`

The set of names of the operations of *type*, as they are defined in the model.

`type.supertypes : Set(OclType)`

The set of all direct supertypes of *type*.
`post: type.allSupertypes->includesAll(result)`

`type.allSupertypes : Set(OclType)`

The transitive closure of the set of all supertypes of *type*.

`type.allInstances : Set(type)`

The set of all instances of *type* and all its subtypes.

OclAny

Within the OCL context, the type OclAny is the supertype of all types in the model. Features on OclAny are available on each object in all OCL expressions.

All classes in a UML model inherit all features defined on OclAny. To avoid name conflicts between features in the model and the features inherited from OclAny, all names on the features of OclAny start with ‘ocl.’ Although theoretically there may still be name conflicts, they can be avoided. One can also use the pathname construct to explicitly refer to the OclAny properties.

Features of OclAny, the instance of OclAny is called *object*.

```
object = (object2 : OclAny) : Boolean
```

True if *object* is the same object as *object2*.

```
object <> (object2 : OclAny) : Boolean
```

True if *object* is a different object from *object2*.

```
post: result = not (object = object2)
```

```
object.oclType : OclType
```

The type of the *object*.

```
object.oclIsKindOf(type : OclType) : Boolean
```

True if *type* is a supertype (transitive) of the type of *object*.

```
post: result = type.allSuperTypes-
>includes(object.oclType) or
               type = object->oclType
```

```
object.oclIsTypeOf(type : OclType) : Boolean
```

True if *type* is equal to the type of *object*.

```
post: result = (object.oclType = type)
```

```
object.oclAsType(type : OclType) : type
```

Results in *object*, but of known type *type*.
Results in Undefined if the actual type of *object* is not type or one of its subtypes.

```
pre : object.oclIsKindOf(type)
post: result = object
post: result.oclIsKindOf(type)
```

OclExpression

Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression. This type and its features are used to define the semantics of features that take an expression as one of their parameters: select, collect, forAll, etc.

An OclExpression includes the optional iterator variable and type and the optional accumulator variable and type.

Features of OclExpression, the instance of OclExpression is called *expression*.

```
expression.evaluationType : OclType
```

The type of the object that results from evaluating *expression*.

Real

The OCL type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

Features of Real, the instance of Real is called *r*.

```
r = (r2 : Real) : Boolean
```

True if *r* is equal to *r2*.

```
r + (r1 : Real) : Real
```

The value of the addition of *r* and *r1*.

```
r - (r1 : Real) : Real
```

The value of the subtraction of *r1* from *r*.

```
r * (r1 : Real) : Real
```

The value of the multiplication of *r* and *r1*.

```
r * (r1 : Real) : Real
```

The value of *r* divided by *r1*.

```
r.abs : Real
```

The absolute value of *r*.

```
post: if r < 0 then result = - r else result = r endif
```

```
r.floor : Integer
```

The largest integer which is less than or equal to *r*.

```
post: (result <= r) and (result + 1 > r)
```

```
r.max(r2 : Real) : Real
```

The maximum of *r* and *r2*.

```
post: if r >= r2 then result = r else result = r2 endif
```

```
r.min(r2 : Real) : Real
```

The minimum of *r* and *r2*.

```
post: if r <= r2 then result = r else result = r2 endif
```

```
r < (r2 : Real) : Boolean
```

True if *r1* is less than *r2*.

`r > (r2 : Real) : Boolean`

True if *r1* is greater than *r2*.
`post: result = not (r <= r2)`

`r <= (r2 : Real) : Boolean`

True if r1 is less than or equal to r2.
`post: result = (r = r2) or (r < r2)`

`r >= (r2 : Real) : Boolean`

True if r1 is greater than or equal to r2.
`post: result = (r = r2) or (r > r2)`

Integer

The OCL type Integer represents the mathematical concept of integer.

Features of Integer, the instance of Integer is called *i*.

`i = (i2 : Integer) : Boolean`

True if *i* is equal to *i2*.

`i + (i2 : Integer) : Integer`

The value of the addition of *i* and *i2*.

`i + (r1 : Real) : Real`

The value of the addition of *i* and *r1*.

`i - (i2 : Integer) : Integer`

The value of the subtraction of *i2* from *i*.

`i - (r1 : Real) : Real`

The value of the subtraction of *r1* from *i*.

`i * (i2 : Integer) : Integer`

The value of the multiplication of *i* and *i2*.

`i * (r1 : Real) : Real`

The value of the multiplication of *i* and *r1*.

`i / (i2 : Integer) : Real`

The value of *i* divided by *i2*.

```
i / (r1 : Real) : Real
```

The value of *i* divided by *r1*.

```
i.abs : Integer
```

The absolute value of *i*.

```
post: if i < 0 then result = - i else result = i endif
```

```
i.div( i2 : Integer) : Integer
```

The number of times that *i2* fits completely within *i*.

```
post: result * i2 <= i
post: result * (i2 + 1) > i
```

```
i.mod( i2 : Integer) : Integer
```

The result is *i* modulo *i2*.

```
post: result = i - (i.div(i2) * i2)
```

```
i.max(i2 : Integer) : Integer
```

The maximum of *i* an *i2*.

```
post: if i >= i2 then result = i else result = i2 endif
```

```
i.min(i2 : Integer) : Integer
```

The minimum of *i* an *i2*.

```
post: if i <= i2 then result = i else result = i2 endif
```

String

The OCL type String represents ASCII strings.

Features of String, the instance of String is called *string*.

```
string = (string2 : String) : Boolean
```

True if *string* and *string2* contain the same characters, in the same order.

```
string.size : Integer
```

The number of characters in *string*.

```
string.concat(string2 : String) : String
```

The concatenation of *string* and *string2*.

```
post: result.size = string.size + string2.size
post: result.substring(1, string.size) = string
post: result.substring(string.size + 1, string2.size) =
string2
```

```
string.toUpper : String
```

The value of *string* with all lowercase characters converted to uppercase characters.

```
post: result.size = string.size
```

```
string.toLower : String
```

The value of *string* with all uppercase characters converted to lowercase characters.

```
post: result.size = string.size
```

```
string.substring(lower : Integer, upper : Integer) : String
```

The sub-string of *string* starting at character number *lower*, up to and including character number *upper.*

Boolean

The OCL type Boolean represents the common true/false values.

Features of Boolean, the instance of Boolean is called *b*.

```
b = (b2 : Boolean) : Boolean
```

Equal if *b* is the same as *b2.*

```
b or (b2 : Boolean) : Boolean
```

True if either *b* or *b2* is true.

```
b xor (b2 : Boolean) : Boolean
```

True if either *b* or *b2* is true, but not both.

```
post: (b or b2) and not (b = b2)
```

```
b and (b2 : Boolean) : Boolean
```

True if both *b1* and *b2* are true.

```
not b : Boolean
```

True if *b* is false.

```
post: if b then result = false else result = true endif
```

```
b implies (b2 : Boolean) : Boolean
```

True if *b* is false, or if *b* is true and *b2* is true.

```
post: (not b) or (b and b2)
```

```
if b then (expression1 : OclExpression)
else (expression2 : OclExpression) endif :
expression1.evaluationType
```

If *b* is true, the result is the value of evaluating *expression1*; otherwise, result is the value of evaluating *expression2*.

Enumeration

The OCL type Enumeration represents the enumerations defined in an UML model.

Features of Enumeration, the instance of Enumeration is called *enumeration*.

```
enumeration = (enumeration2 : Boolean) : Boolean
```

Equal if *enumeration* is the same as *enumeration2*.

```
enumeration <> (enumeration2 : Boolean) : Boolean
```

Equal if *enumeration* is not the same as *enumeration2*.

```
post: result = not ( enumeration = enumeration2)
```

B.7.2 Collection-Related Typed

The following sections define the features on collections (i.e., these features are available on Set, Bag, and Sequence). As defined in this section, each collection type is actually a template with one parameter. ‘T’ denotes the parameter. A real collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

Collection

Collection is the abstract supertype of all collection types in OCL. Each occurrence of an object in a collection is called an element. If an object occurs twice in a collection, there are two elements. This section defines the operations on Collections that have identical semantics for all collection subtypes. Some operations may be defined with the subtype as well, which means that there is an additional postcondition or a more specialized return value.

The definition of several common operations is different for each subtype. These operations are not mentioned in this section.

Features of Collection, the instance of Collection is called *collection*.

```
collection->size : Integer
```

The number of elements in the collection *collection*.

```
post: result = collection->iterate(elem; acc : Integer = 0 | acc
+ 1)
```

```
collection->includes(object : OclAny) : Boolean
```

True if *object* is an element of *collection*, false otherwise.

```
post: result = (collection->count(object) > 0)
```

```
collection->count(object : OclAny) : Integer
```

The number of times that *object* occurs in the collection *collection.*

```
post: result = collection->iterate( elem; acc : Integer = 0 |
                 if elem = object then acc + 1 else acc endif)
```

```
collection->includesAll(c2 : Collection(T)) : Boolean
```

Does *collection* contain all the elements of *c2* ?

```
post: result = c2->forAll(elem | collection->includes(elem))
```

```
collection->isEmpty : Boolean
```

Is *collection* the empty collection?

```
post: result = ( collection->size = 0 )
```

```
collection->notEmpty : Boolean
```

Is *collection* not the empty collection?

```
post: result = ( collection->size <> 0 )
```

```
collection->sum : T
```

The addition of all elements in *collection.* Elements must be of a type supporting addition (Integer and Real)

```
post: result = collection->iterate( elem; acc : T = 0 |
                        acc + elem )
```

```
collection->exists(expr : OclExpression) : Boolean
```

Results in true if *expr* evaluates to true for at least one element in *collection.*

```
post: result = collection->iterate(elem; acc : Boolean = false |
                 acc or expr)
```

```
collection->forAll(expr : OclExpression) : Boolean
```

Results in true if *expr* evaluates to true for each element in *collection*; otherwise, result is false.

```
post: result = collection->iterate(elem; acc : Boolean = true |
                 acc and expr)
```

```
collection->iterate(expr : OclExpression) : expr.evaluationType
```

Iterates over the collection. See “Iterate Operation” on page B-22 for a complete description. This is the basic collection operation with which the other collection operations can be described.

Set

The Set is the mathematical set. It contains elements without duplicates. Features of Set, the instance of Set is called *set*.

```
set->union(set2 : Set(T)) : Set(T)
```

The union of *set* and *set2*.

```
post: T.allInstances->forAll(elem |
         result->includes(elem) =
              set->includes(elem) or set2->includes(elem)
```

```
set->union(bag : Bag(T)) : Bag(T)
```

The union of *set* and *bag*.

```
post: T.allInstances->forAll(elem |
         result->count(elem) =
              set->count(elem) + bag->count(elem))
```

```
set = (set2 : Set) : Boolean
```

Evaluates to true if *set* and *set2* contain the same elements.

```
post: result = T.allInstances->forAll(elem |
              set->includes(elem) = set2->includes(elem))
```

```
set->intersection(set2 : Set(T)) : Set(T)
```

The intersection of *set* and *set2* (i.e, the set of all elements that are in both *set* and *set2*).

```
post: T.allInstances->forAll(elem |
         result->includes(elem) =
             set->includes(elem) and set2->includes(elem))
```

```
set->intersection(bag : Bag(T)) : Set(T)
```

The intersection of *set* and *bag*.

```
post: result = set->intersection( bag->asSet )
```

```
set – (set2 : Set(T)) : Set(T)
```

The elements of *set*, which are not in *set2*.

```
post: T.allInstances->forAll(elem |
         result->includes(elem) =
                set->includes(elem) and not set2-
              >includes(elem))
```

```
set->including(object : T) : Set(T)
```

The set containing all elements of *set* plus *object*.

```
post: T.allInstances->forAll(elem |
         result->includes(elem) =
               set->includes(elem) or (elem = object))
```

```
set->excluding(object : T) : Set(T)
```

The set containing all elements of *set* without *object*.

```
post: T.allInstances->forAll(elem |
         result->includes(elem) =
               set->includes(elem) and not(elem = object))
```

```
set->symmetricDifference(set2 : Set(T)) : Set(T)
```

The sets containing all the elements that are in *set* or *set2*, but not in both.

```
post: T.allInstances->forAll(elem |
        result->includes(elem) =
            set->includes(elem) xor set2->includes(elem))
```

```
set->select(expr : OclExpression) : Set(expr.type)
```

The subset of *set* for which *expr* is true.

```
post: result = set->iterate(elem; acc : Set(T) = Set{} |
         if expr then acc->including(elem) else acc endif)
```

```
set->reject(expr : OclExpression) : Set(expr.type)
```

The subset of *set* for which *expr* is false.

```
post: result = set->select(not expr)
```

```
set->collect(expression : OclExpression) :
Bag(expression.oclType)
```

The Bag of elements which results from applying *expr* to every member of *set*.

```
post: result = set->iterate(elem; acc : Bag(T) = Bag{} |
               acc->including(expr) )
```

```
set->count(object : T) : Integer
```

The number of occurrences of *object* in *set*.

```
post: result <= 1
```

```
set->asSequence : Sequence(T)
```

A Sequence that contains all the elements from *set*, in random order.

```
post: T.allInstances->forAll(elem |
       result->count(elem) = set->count(elem))
```

```
set->asBag : Bag(T)
```

The Bag that contains all the elements from *set*.

```
post: T.allInstances->forAll(elem |
       result->count(elem) = set->count(elem))
```

Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag. Features of Bag, the instance of Bag is called *bag*.

```
bag = (bag2 : Bag) :  Boolean
```

True if *bag* and *bag2* contain the same elements, the same number of times.

```
post: result = T.allInstances->forAll(elem |
                    bag->count(elem) = bag2->count(elem))
```

```
bag->union(bag2 : Bag) : Bag(T)
```

The union of *bag* and *bag2*.

```
post: T.allInstances->forAll(elem |
         result->count(elem) =
              bag->count(elem) + bag2->count(elem))
```

```
bag->union(set : Set) : Bag(T)
```

The union of *bag* and *set*.

```
post: T.allInstances->forAll(elem |
         result->count(elem) =
              bag->count(elem) + set->count(elem))
```

```
bag->intersection(bag2 : Bag) : Bag(T)
```

The intersection of *bag* and *bag2*.

```
post: T.allInstances->forAll(elem |
         result->count(elem) =
              bag->count(elem).min(bag2->count(elem)) )
```

```
bag->intersection(set : Set) : Set(T)
```

The intersection of *bag* and *set*.

```
post: T.allInstances->forAll(elem |
         result->count(elem) =
              bag->count(elem).min(set->count(elem)) )
```

```
bag->including(object : T) : Bag(T)
```

The bag containing all elements of *bag* plus *object*.

```
post: T.allInstances->forAll(elem |
     if elem = object then
          result->count(elem) = bag->count(elem) + 1
     else
          result->count(elem) = bag->count(elem)
     endif)
```

```
bag->excluding(object : T) : Bag(T)
```

The bag containing all elements of bag apart from all occurrences of *object*.

```
post: T.allInstances->forAll(elem |
     if elem = object then
          result->count(elem) = 0
     else
          result->count(elem) = bag->count(elem)
     endif)
```

```
bag->select(expression : OclExpression) : Bag(T)
```

The sub-bag of *bag* for which *expression* is true.

```
post: result = bag->iterate(elem; acc : Bag(T) = Bag{} |
          if expr then acc->including(elem) else acc endif)
```

```
bag->reject(expression : OclExpression) : Bag(T)
```

The sub-bag of *bag* for which *expression* is false.

```
post: result = bag->select(not expr)
```

```
bag->collect(expression: OclExpression) :
Bag(expression.oclType)
```

The Bag of elements which results from applying *expression* to every member of *bag*.

```
post: result = bag->iterate(elem; acc : Bag(T) = Bag{} |
                    acc->including(expr) )
```

```
bag->count(object : T) : Integer
```

The number of occurrences of *object* in *bag*.

```
bag->asSequence : Sequence(T)
```

A Sequence that contains all the elements from *bag*, in random order.

```
post: T.allInstances->forAll(elem |
                    bag->count(elem) = result->count(elem))
```

```
bag->asSet : Set(T)
```

The Set containing all the elements from *bag*, with duplicates removed.

```
post: T.allInstances(elem |
            bag->includes(elem) = result->includes(elem))
```

Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Features of Sequence(T), the instance of Sequence is called *sequence*.

```
sequence->count(object : T) : Integer
```

The number of occurrences of *object* in *sequence*.

```
sequence = (sequence2 : Sequence(T)) : Boolean
```

True if *sequence* contains the same elements as *sequence2* in the same order.

```
post: result = Sequence{1..sequence->size}->forAll(index :
Integer |
           sequence->at(index) = sequence2->at(index))
          and
          sequence->size = sequence2->size
```

```
sequence->union (sequence2 : Sequence(T)) : Sequence(T)
```

The sequence consisting of all elements in *sequence*, followed by all elements in *sequence2*.

```
post: result->size = sequence->size + sequence2->size
post: Sequence{1..sequence->size}->forAll(index : Integer |
         sequence->at(index) = result->at(index))
post: Sequence{1..sequence2->size}->forAll(index : Integer |
         sequence2->at(index) =
                        result->at(index + sequence->size)))
```

```
sequence->append (object: T) : Sequence(T)
```

The sequence of elements, consisting of all elements of *sequence*, followed by *object*.

```
post: result->size = sequence->size + 1
post: result->at(result->size) = object
post:   Sequence{1..sequence->size}->forAll(index : Integer |
          result->at(index) = sequence ->at(index))
```

```
sequence->prepend(object : T) : Sequence(T)
```

The sequence consisting of all elements in *sequence*, followed by *object*.

```
post: result->size = sequence->size + 1
post: result->at(1) = object
post:   Sequence{1..sequence->size}->forAll(index : Integer |
         sequence->at(index) = result->at(index + 1))
```

`sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)`

The sub-sequence of *sequence* starting at number *lower*, up to and including element number *upper*.

```
post: if sequence->size < upper then
          result = Undefined
else
          result->size = upper - lower + 1 and
           Sequence{lower..upper}->forAll( index |
          result->at(index - lower + 1) =
                                 sequence->at(lower + index - 1))
endif
```

`sequence->at(i : Integer) : T`

The *i-th* element of sequence.

```
post: i <= 0 or sequence->size < i implies result =
Undefined
```

`sequence->first : T`

The first element in *sequence*.

```
post: result = sequence->at(1)
```

`sequence->last : T`

The last element in *sequence*.

```
post: result = sequence->at(sequence->size)
```

`sequence->including(object : T) : Sequence(T)`

The sequence containing all elements of *sequence* plus *object* added as the last element.

```
post: result = sequence.append(object)
```

`sequence->excluding(object : T) : Sequence(T)`

The sequence containing all elements of *sequence* apart from all occurrences of *object*. The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size = sequence->size - sequence->count(object)
post: result = sequence->iterate(elem; acc : Sequence(T)
      = Sequence{}|
          if elem = object then acc else acc->append(elem) endif )
```

`sequence->select(expression : OclExpression) : Sequence(T)`

The subsequence of *sequence* for which *expression* is *true*.

```
post: result = sequence->iterate(elem; acc : Sequence(T) =
Sequence{} |
          if expr then acc->including(elem) else acc endif)
```

```
sequence->reject(expression : OclExpression) : Sequence(T)
```

The subsequence of *sequence* for which *expression* is false.

```
post: result = sequence->select(not expr)
```

```
sequence->collect(expression : OclExpression) :

Sequence(expression.oclType)
```

The Sequence of elements which results from applying *expression* to every member of *sequence*.

```
sequence->iterate(expr : OclExpression) : expr.evaluationType
```

Iterates over the sequence. Iteration will be done from element at position 1 up until the element at the last position following the order of the sequence.

```
sequence->asBag() : Bag(T)
```

The Bag containing all the elements from *sequence*, including duplicates.

```
post: T.allInstances->forAll(elem |
        result->count(elem) = sequence->count(elem))
```

```
sequence->asSet() : Set(T)
```

The Set containing all the elements from *sequence*, with duplicated removed.

```
post: T.allInstances->forAll(elem |
        result->includes(elem) = sequence->includes(elem))
```

B.11 Grammar for OCL

This section describes the grammar for OCL expressions. An executable LL(1) version of this grammar is available on the OCL web site. (See http://www.software.ibm.com/ad/ocl).

The grammar description uses the EBNF syntax, where "|" means a choice, "?" optionality, and "*" means zero or more times. In the description of the *name*, *typeName*, and *string*, the syntax for lexical tokens from the JavaCC parser generator is used. (See http://www.suntest.com/JavaCC.)

```
expression                  := logicalExpression
ifExpression                := "if" expression
                               "then" expression
                               "else" expression
                               "endif"
logicalExpression           := relationalExpression
```

```
                               ( logicalOperator
                                 relationalExpression )*
relationalExpression      := additiveExpression
                              ( relationalOperator
                                additiveExpression )?
additiveExpression        := multiplicative Expression
                              ( addOperator
                                multiplicativeExpression )*
multiplicativeExpressin   := unaryExpression
                              ( multiplyOperator unaryExpression )*
unaryExpression           :=  ( unaryOperator postfixExpression )
                              | postfixExpression
postfixExpression         := primaryExpression ( ("." | "->")
                            featureCall )*
primaryExpression         := literalCollection
                              | literal
                              | pathName timeExpression? qualifier?
                           featureCallParameters?
                              | "(" expression ")"
                              | ifExpression
featureCallParameters     := "(" ( declarator )?
                             ( actualParameterList )? ")"
literal                   := <STRING> | <number> | "#" <name>
enumerationType           := "enum" "{" "#" <name> ( "," "#" <name>
                                     )* "}"
simpleTypeSpecifier       := pathTypeName
                              | enumerationType
literalCollection         := collectionKind "{"
expressionListOrRange? "}"
expressionListOrRange     := expression
                               ( ( "," expression )+
                              | ( ".." expression )
                               )?
featureCall               := pathName timeExpression? qualifiers?
                               featureCallParameters?
qualifiers                := "[" actualParameterList "]"
declarator                := <name> ( "," <name> )*
                               ( ":" simpleTypeSpecifier )? "|"
pathTypeName              := <typeName> ( "::" <typeName> )*
pathName                  := ( <typeName> | <name> )
                                ( "::" ( <typeName> | <name> ) )*
timeExpression            := "@" <name>
```

```
actualParameterList      := expression ( "," expression )*
logicalOperator          := "and" | "or" | "xor" | "implies"
collectionKind           := "Set" | "Bag" | "Sequence" |
                               "Collection"
relationalOperator       := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator              := "+" | "-"
multiplyOperator         := "*" | "/"
unaryOperator            := "-" | "not"
typeName                 := "A"-"Z" ( "a"-"z" | "0"-"9" | "A"-"Z"
                                    | "_")*
name                     := "a"-"z" ( "a"-"z" | "0"-"9" | "A"-"Z"
                                    | "_")*
number                   := "0"-"9" ("0"-"9")*
string                   := "'" ( (~["'","\\","\n","\r"])
                                    | ("\\"
                           ( ["n","t","b","r","f","\\","'","\""]
                                    | ["0"-"7"] ( ["0"-"7"] )?
                                    | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                                             )
                                            )
                                          )*
                                       "'"
```

Index

Index

Index