Date: November 2008

Business Process Definition MetaModel Volume I: Common Infrastructure

Version 1.0

OMG Document Number: formal/2008-11-03
Standard document URL: http://www.omg.org/spec/BPDM/1.0
Associated File(s)*: http://www.omg.org/spec/BPDM/20080501

http://www.omg.org/spec/BPDM/20080501/Abstractions.xsd
http://www.omg.org/spec/BPDM/20080501/Activity.xsd
http://www.omg.org/spec/BPDM/20080501/BehaviorModel.xsd
http://www.omg.org/spec/BPDM/20080501/bpdm.xsd
http://www.omg.org/spec/BPDM/20080501/bpmn.cmof
http://www.omg.org/spec/BPDM/20080501/BPMNLibrary
http://www.omg.org/spec/BPDM/20080501/CommonInfrastructure.cmof
http://www.omg.org/spec/BPDM/20080501/CommonInfrastructureLibrary
http://www.omg.org/spec/BPDM/20080501/CompositionModel.xsd
http://www.omg.org/spec/BPDM/20080501/ConditionModel.xsd
http://www.omg.org/spec/BPDM/20080501/CourseModel.xsd
http://www.omg.org/spec/BPDM/20080501/importfile_commoninfrastructure.xsd
http://www.omg.org/spec/BPDM/20080501/InteractionProtocol.xsd
http://www.omg.org/spec/BPDM/20080501/InteractiveBehaviorModel.xsd
http://www.omg.org/spec/BPDM/20080501/xmi_infra.xsd
http://www.omg.org/spec/BPDM/20080501/VotingSample
http://www.omg.org/spec/BPDM/20080501/BPMNSamples_schema.xsd

http://www.omg.org/spec/BPDM/20080502
http://www.omg.org/spec/BPDM/20080502/xmi.xsd

Source document: BPDM Common Infrastructure Document without change bars (dtc/2008-05-07)
* Original file: XML schema and library (dtc/2008-05-14)

OMG's Issue Reporting Procedure

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

Table of Contents

List of Figures

1 Normative References

[OMG formal/2007-11-04] http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

2 Terms and Definitions

Classifier

A classifier is a classification of instances - it describes a set of instances that have features in common.

Description : A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

Element

An element can own comments. The comments for an Element add no semantics but may represent information useful to the reader of the model.

DataType

DataType is an abstract class that acts as a common superclass for different kinds of data types. DataType is the abstract class that represents the general notion of being a data type (i.e., a type whose instances are identified only by their value).

Expression

An expression is a structured tree of symbols that denotes a (possibly empty) set of values when evaluated in a context. An expression represents a node in an expression tree, which may be non-terminal or terminal. It defines a symbol, and has a possibly empty sequence of operands that are value specifications.

ValueSpecification

A value specification is the specification of a (possibly empty) set of instances, including both objects and data values.

Description: ValueSpecification is an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated.

Generalization

A generalization between two types means each instance of the specific type is also an instance of the general type. Any specification applying to instances of the general type also apply to instances of the specific type.

Namespace

A namespace is a named element that can own other named elements. Each named element may be owned by at most one namespace. A namespace provides a means for identifying named elements by name. Named elements can be identified by name in a namespace either by being directly owned by the namespace or by being introduced into the namespace by other means (e.g., importing or inheriting). Namespace is an abstract metaclass.

Package

A package is a container for types and other packages. Packages provide a way of grouping types and packages together, which can be useful for understanding and managing a model. A package cannot contain itself.

Property

A property is a structural feature of a classifier that characterizes instances of the classifier.

Description: Property represents a declared state of one or more instances in terms of a named relationship to a value or values. When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance. Property is indirectly a subclass of TypedElement. The range of valid values represented by the property can be controlled by setting the property's type.

Type

A **Type** is a **NamedElement** that groups individuals according to some commonality among them, which might be characteristics they can have or constraints they obey. Types can cover any kind of entity, physical or computational, static or dynamic. For example, the type Person groups individual people, like Mary and John. The type declares commonalities among people, for example, they can have names and gender, or obey constraints, such as being genetically related to exactly two other people.

TypedElement

A typed element is a kind of named element that represents elements with types. Elements with types are instances of TypedElement. A typed element may optionally have no type. The type of a typed element constrains the set of values that the typed element may refer to.

Composite

A **Composite** is a **Classifier** which has an internal structure. It specifies the connections of individuals that are all related to the same other individual (M0). For example, a company type specifies the connections of departments within each individual company of that type (assuming it is modeled in a value chain manner, rather than just an organization chart). Likewise, an orchestration type specifies the sequence of steps in each individual occurrence of that orchestration.

Part

A **Part** is a **Connectable Element** that is an element of the structure of a **Composite**.

Part Connection

A **Part Connection** is a **Feature** of a composite used to connect its **Connectable Elements**. A **Part Connection** can connect any number of parts. For example, a business interaction can involve multiple companies.

When a **Part Connection** is connecting **Typed Part**, its specifies links between M0 entities playing the typed parts. For example, the reporting connection between the president of a company and the CEO means the person playing the president in a particular company will report to the person playing the CEO in the same company. Likewise, the temporal connection between one step and another in a process means that in each occurrence of that process, there is an occurrence of one step that happens after the occurrence of another. **Conditions** may be applied to **Part Connections** to limit when they apply. For example, one step in a process may happen after another only when certain conditions are true as the process is executing.

Condition

A **Condition** is a **Boolean ValueSpecification** that constrains some element in the models. Conditions are true if their descriptions hold in the current state of the world, possibly including executions, and false otherwise.

Statement

Statement is a **Boolean ValueSpecification** that does not constrain anything. **Statement**s are used to integrate with rule models.

Course

A **Course** is an ordered **Succession** of **Happening Part**s. A **Course** is a **Composite** that has connections representing that one part of the course "follows" another in time, and possibly establishes constraints on such followings (**Succession**).

Course Part

A **Course Part** is a kind of **Connectable Element** that defines a stage in a **Course**. It can be connected to **Succession** as a **predecessor** or **successor** element.

Event

An **Event** is a **Happening** for dynamic entities occurring at a point in time.

Event Condition

An **Event Condition** is a **Condition** for specifying that an **Event** must occur in the context of a particular **Happening Over Time** for the condition to hold. For instance, a condition can be on the eruption (instance of **Event**) of a particular volcano (instance of **Happening Over Time**).

Event Part

An **Event Part** identifies **Event** (such as **Start Event** or **End Event**) for an individual **Course**. An **Event Part** is also a **Happening Part**, enabling it to be connected by **Successions**.

Gateway

A **Gateway** is a kind of **Course Part** representing potentially complex specifications of how dynamic individuals playing **Happening Part**s are ordered in time. The particular specifications are given in subtypes. At runtime, **Gateways** don't have any execution trace.

Happening

A **Happening** is a **Classifier** for dynamic entities.

Succession

A **Succession** is a **Directed Part Connection** that organizes **Course Parts** in series in the context of a **Course**. A **Succession** indicates that one **Course Part** "follows" another in time, and possibly establishes constraints on such followings. It can order the **Event Part** of its **Happening Parts** such as their **Start** or **End**.
Succession allows any combination of **Event Part** to be connected.

End -> Start
Start -> Start
Start -> Abort
etc.

A **Succession** doesn't need to have **Happening Part** on its ends, it can have untyped course parts also, such as **Gateway**, but it must have something on each end. For convenience, a **Succession** that does not specify **source event part** or **target event part** will have the same effect as a **Succession** where these are respectively the **End** and **Start**.

Time Event

A **Time Event** specifies a point in time that is a source of interest.

Time Event Condition

A **Time Event Condition** is a kind of **Event Condition** that is based on the occurrence of a **Time Event**. A **Time Event Condition** is specified by referring to a **Clock**.

3 Additional Information

3.1 Acknowledgements

The following companies submitted this specification:

- Adaptive
- Axway Software
- Borland Software
- Model Driven Solutions
- EDS
- Lombardi Software
- MEGA International
- Unisys

The following companies and organizations support this specification:

- BPM Focus
- U.S. National Institute of Standards and Technology (NIST)

4 Metamodel and Notation Specification

This section presents the normative specification for the common infrastructure metamodel. It begins with an overview of the metamodel structure followed by a description of each sub-package.

4.1 Overview

The **Abstractions** package is a result of the merge from the InfrastructureLibrary::Core:Abstractions package and the Infrastructure:Core:PrimitiveTypes package.

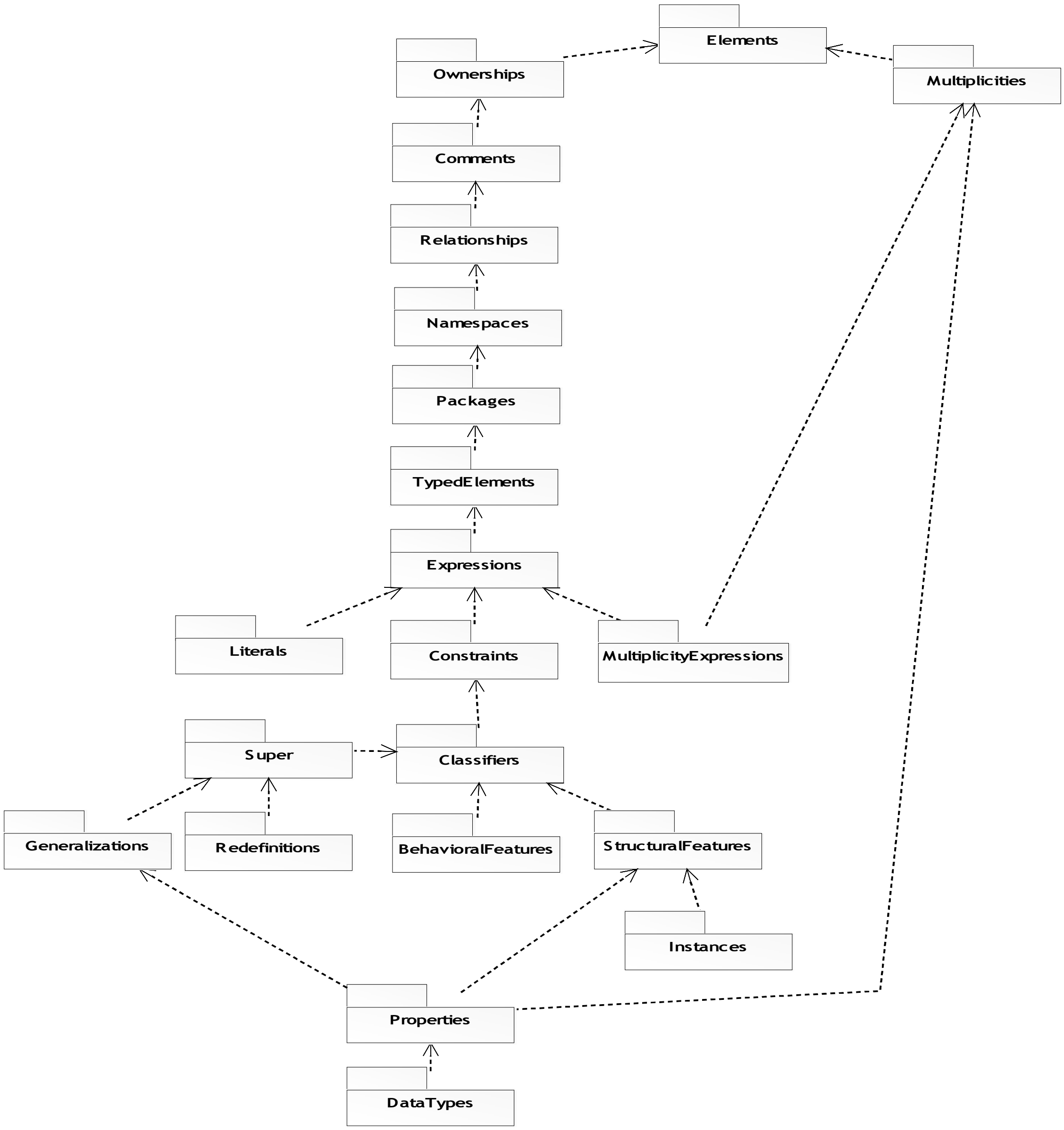


Figure 1 - Package Dependencies

Package	**Comment**
BehavioralFeatures	The BehavioralFeatures subpackage of the Abstractions package specifies the basic classes for modeling dynamic features of model elements.
Classifiers	The Classifiers package in the Abstractions package specifies an abstract generalization for the classification of instances according to their features.
Comments	The Comments package of the Abstractions package defines the general capability of attaching comments to any element.
Constraints	The Constraints subpackage of the Abstractions package specifies the basic building blocks that can be used to add additional semantic information to an element.

DataTypes	The DataTypes subPackage specifies the DataType, Enumeration, EnumerationLiteral, and PrimitiveType constructs. These constructs are used for defining primitive data types (such as Integer and String) and user-defined enumeration data types. The data types are typically used for declaring the types of the class attributes.
Elements	The Elements subpackage of the Abstractions package specifies the most basic abstract construct, Element.
Expressions	The Expressions package in the Abstractions package specifies the general metaclass supporting the specification of values, along with specializations for supporting structured expression trees and opaque, or uninterpreted, expressions. Various UML constructs require or use expressions, which are linguistic formulas that yield values when evaluated in a context.
Generalizations	The Generalizations package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.
Instances	The Instances package in the Abstractions package provides for modeling instances of classifiers.
Literals	The Literals package in the Abstractions package specifies metaclasses for specifying literal values.
Multiplicities	The Multiplicities subpackage of the Abstractions package defines the metamodel classes used to support the specification of multiplicities for typed elements (such as association ends and attributes), and for specifying whether multivalued elements are ordered or unique.
MultiplicityExpressions	The MultiplicityExpressions subpackage of the Abstractions package extends the multiplicity capabilities to support the use of value expressions for the bounds.
Namespaces	The Namespaces subpackage of the Abstractions package specifies the concepts used for defining model elements that have names, and the containment and identification of these named elements within namespaces.
Ownerships	The Ownerships subpackage of the Abstractions package extends the basic element to support ownership of other elements.
Packages	The Packages package of Abstractions specifies the Package and PackageImport constructs.
Properties	The Properties subpackage of the Abstractions package specifies the basic class for modeling structural features of model elements.
Redefinitions	
Relationships	The Relationships subpackage of the Abstractions package adds support for directed relationships.
StructuralFeatures	The StructuralFeatures package of the Abstractions package specifies an abstract generalization of structural features of classifiers.
Super	The Super package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.
TypedElements	The TypedElements subpackage of the Abstractions package defines typed elements and their types.

4.2 Abstractions

4.2.1 Introduction

The Abstractions package represents the core modeling concepts of the UML, including classifiers, properties, and packages. This part is mostly reused from the infrastructure library, since many of these concepts are the same as those that are used in, for example, MOF.

4.2.2 Metamodel

The PrimitiveTypes package of InfrastructureLibrary::Core contains a number of predefined types used when defining the abstract syntax of metamodels.

4.2.2.1 PrimitiveTypes

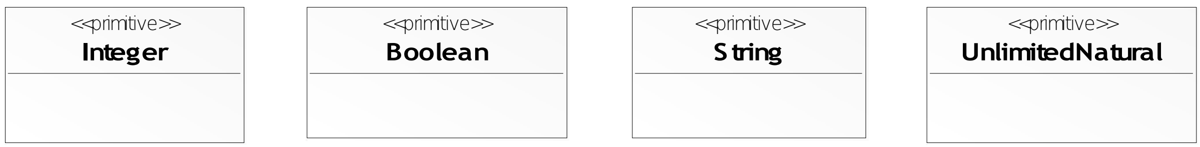


Figure 2 - Primitive Types

4.2.2.2 Boolean

Package: PrimitiveTypes
isAbstract: No

Description

Boolean is an instance of PrimitiveType. In the metamodel, Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

- true - The Boolean condition is satisfied.
- false - The Boolean condition is not satisfied.

4.2.2.3 Integer

Package: PrimitiveTypes
isAbstract: No

Description

An instance of Integer is an element in the (infinite) set of integers (..2, -1, 0, 1, 2..). It is used for integer attributes and integer expressions in the metamodel.

4.2.2.4 String

Package: PrimitiveTypes
isAbstract: No

Description

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.

An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

4.2.2.5 UnlimitedNatural

Package: PrimitiveTypes
isAbstract: No

Description

An unlimited natural is a primitive type representing unlimited natural values. An instance of UnlimitedNatural is an element in the (infinite) set of naturals (0, 1, 2..). The value of infinity is shown using an asterisk ('*').
The Elements subpackage of the Abstractions package specifies the most basic abstract construct, Element.

4.2.2.6 Elements Package

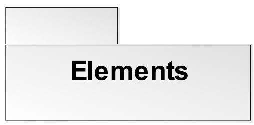


Figure 3 - Elements Package

4.2.2.7 Elements

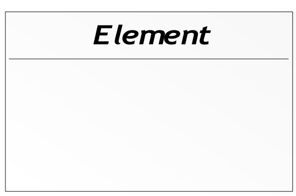


Figure 4 - Elements

4.2.2.8 Element

Package: Elements
isAbstract: Yes

Description

An element is a constituent of a model.

Description
Element is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library.

The Ownerships subpackage of the Abstractions package extends the basic element to support ownership of other elements.

4.2.2.9 Ownerships Package

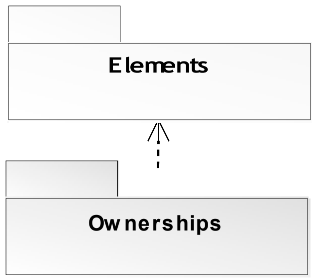


Figure 5 - Ownerships Package

4.2.2.10 Ownerships

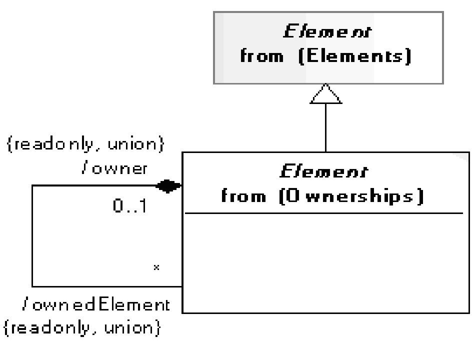


Figure 6 - Ownerships

4.2.2.11 Element

Package: Ownerships
isAbstract: Yes
Generalization: "Element"

Description

An element is a constituent of a model. As such, it has the capability of owning other elements.

Description
Element has a derived composition association to itself to support the general capability for elements to own other elements.

The Comments package of the Abstractions package defines the general capability of attaching comments to any element.

4.2.2.12 Comments Package

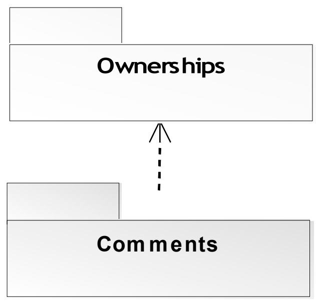


Figure 7 - Comments Package

4.2.2.13 Comments

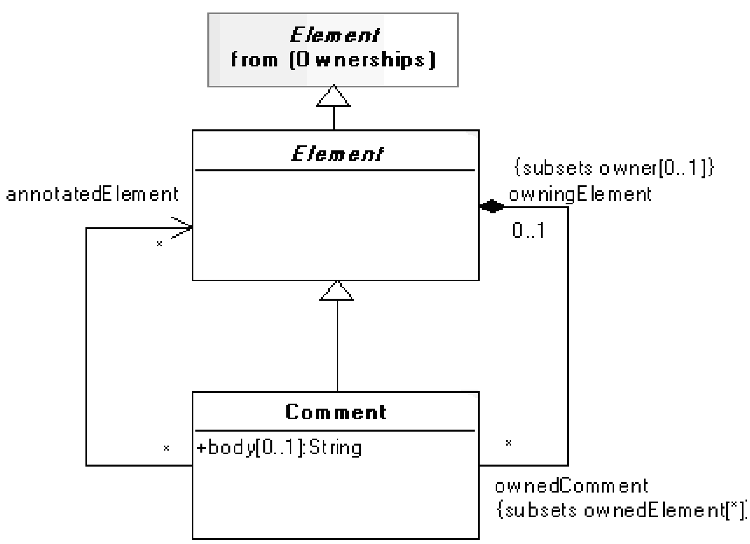


Figure 8 - Comments

4.2.2.14 Comment

Package: Comments
isAbstract: No
Generalization: “Element”

Description

A comment is a textual annotation that can be attached to a set of elements. A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler. A comment may be owned by any element. A Comment adds no semantics to the annotated elements, but may represent information useful to the reader of the model.

Attributes

body: String [0..1]	Specifies a string that is the comment.

Associations

annotatedElement : Element [*]	References the Element(s) being commented.

4.2.2.15 Element

Package: Comments
isAbstract: Yes
Generalization: "Element"

Description

An element can own comments. The comments for an Element add no semantics but may represent information useful to the reader of the model.

Associations

ownedComment : Comment [*]	The Comments owned by this element. Subsets *ownedElement*

The Relationships subpackage of the Abstractions package adds support for directed relationships.

4.2.2.16 Relationships Package

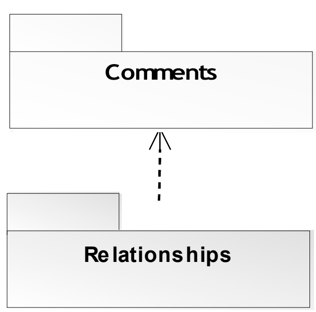


Figure 9 - Relationships Package

4.2.2.17 Relationships

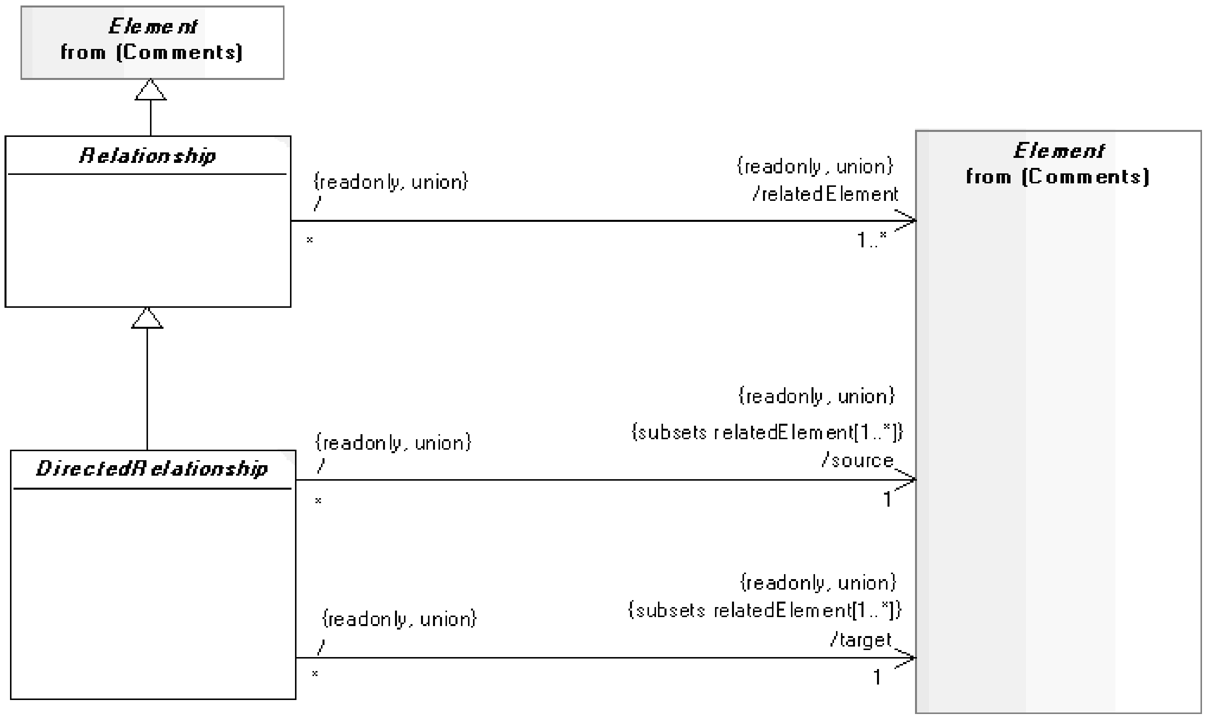


Figure 10 - Relationships

4.2.2.18 DirectedRelationship

Package: Relationships
isAbstract: Yes
Generalization: "Relationship"

Description

A directed relationship represents a relationship between a collection of source model elements and a collection of target model elements.

Associations

source : Element [1]	Specifies the sources of the DirectedRelationship. This is a derived union. Subsets *relatedElement*
target : Element [1]	Specifies the targets of the DirectedRelationship. This is a derived union. Subsets *relatedElement*

4.2.2.19 Relationship

Package: Relationships
isAbstract: Yes
Generalization: "Element"

Description

Relationship is an abstract concept that specifies some kind of relationship between elements.

Associations

relatedElement : Element [1..*]	Specifies the elements related by the Relationship. This is a derived union.

The Namespaces subpackage of the Abstractions package specifies the concepts used for defining model elements that have names, and the containment and identification of these named elements within namespaces.

4.2.2.20 Namespaces Package

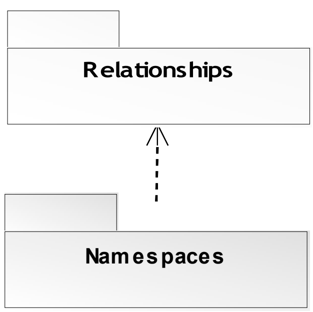


Figure 11 - Namespaces Package

4.2.2.21 Namespaces

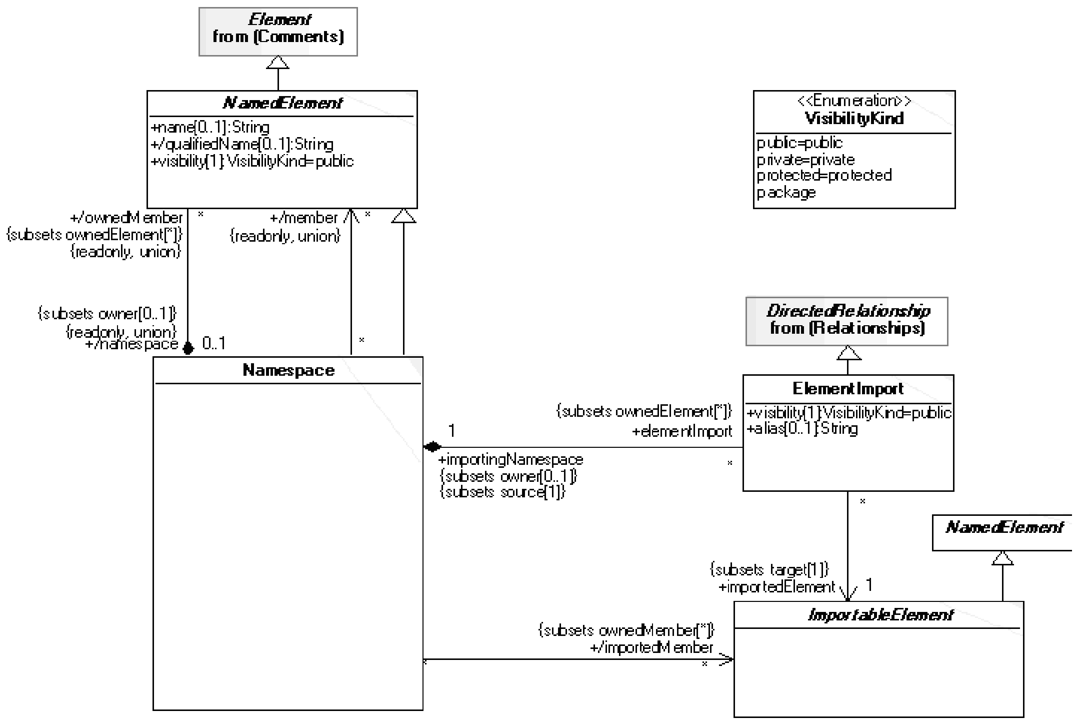


Figure 12 - Namespaces

4.2.2.22 ElementImport

Package: Namespaces
isAbstract: No
Generalization: “DirectedRelationship”

Description

An element import identifies an element in another package, and allows the element to be referenced using its name without a qualifier.

Description
An element import is defined as a directed relationship between an importing namespace and a packageable element. The name of the packageable element or its alias is to be added to the namespace of the importing namespace. It is also possible to control whether the imported element can be further imported.

Semantics
An element import adds the name of a packageable element from a package to the importing namespace. It works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported. An element import is used to selectively import individual elements without relying on a package import. In case of a nameclash with an outer name (an element that is defined in an enclosing namespace is available using its unqualified name in enclosed namespaces) in the importing namespace, the outer name is hidden by an element import, and the unqualified name refers to the imported element. The outer name can be accessed using its qualified name.

If more than one element with the same name would be imported to a namespace as a consequence of element imports or package imports, the elements are not added to the importing namespace and the names of those elements must be qualified in order to be used in that namespace. If the name of an imported element is the same as the name of an element owned by the importing namespace, that element is not added to the importing namespace and the name of that element must be qualified in order to be used. If the name of an imported element is the same as the name of an element owned by the importing namespace, the name of the imported element must be qualified in order to be used and is not added to the importing namespace.

An imported element can be further imported by other namespaces using either element or package imports.
The visibility of the ElementImport may be either the same or more restricted than that of the imported element.

Attributes

visibility: VisibilityKind [1]	Specifies the visibility of the imported ImportableElement within the importing Namespace. The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import; default value is public.
alias: String [0..1]	Specifies the name that should be added to the namespace of the importing Package in lieu of the name of the imported PackagableElement. The aliased name must not clash with any other member name in the importing Package. By default, no alias is used.

Associations

importedElement : ImportableElement [1]	Specifies the PackageableElement whose name is to be added to a Namespace. Subsets *target*

4.2.2.23 ImportableElement

Package: Namespaces
isAbstract: Yes
Generalization: “NamedElement”

Description

A **ImportableElement** indicates a named element that is imported by a **Namespace**.

4.2.2.24 NamedElement

Package: Namespaces
isAbstract: Yes
Generalization: “Element”

Description

A named element represents elements with names. Elements with names are instances of NamedElement. The name for a named element is optional. If specified, then any valid string, including the empty string, may be used.

Attributes

name: String [0..1]	The name of the NamedElement.
qualifiedName: String [0..1]	A name which allows the NamedElement to be identified within a hierarchy of nested Namespaces. It is constructed from the names of the containing

	namespaces starting at the root of the hierarchy and ending with the name of the NamedElement itself. This is a derived attribute.
visibility: VisibilityKind [1]	Determines where the NamedElement appears within different Namespaces within the overall model, and its accessibility.

4.2.2.25 Namespace

Package: Namespaces
isAbstract: Yes
Generalization: "NamedElement"

Description

A namespace is a named element that can own other named elements. Each named element may be owned by at most one namespace. A namespace provides a means for identifying named elements by name. Named elements can be identified by name in a namespace either by being directly owned by the namespace or by being introduced into the namespace by other means (e.g., importing or inheriting). Namespace is an abstract metaclass.

Associations

elementImport : ElementImport [*]	References the ElementImports owned by the Namespace. Subsets *ownedElement* Subsets
importedMember : ImportableElement [*]	References the ImportableElements that are members of this Namespace as a result of either ElementImports. This is a derived association. Subsets *ownedMember*
member : NamedElement [*]	A collection of NamedElements identifiable within the Namespace, either by being owned or by being introduced by importing or inheritance. This is a derived union.
ownedMember : NamedElement [*]	A collection of NamedElements owned by the Namespace. This is a derived union. Subsets *ownedElement*

4.2.2.26 VisibilityKind

Package: Namespaces
isAbstract: No

Description

VisibilityKind is an enumeration type that defines literals to determine the visibility of elements in a model.

Semantics

VisibilityKind is intended for use in the specification of visibility in conjunction with, for example, the Imports, Generalizations, Packages, and Classifiers packages. Detailed semantics are specified with those mechanisms. If the Visibility package is used without those packages, these literals will have different meanings, or no meanings.

- A public element is visible to all elements that can access the contents of the namespace that owns it.
- A private element is only visible inside the namespace that owns it.

- A protected element is visible to elements that have a generalization relationship to the namespace that owns it.
- A package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace.

Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible.

In circumstances where a named element ends up with multiple visibilities, for example by being imported multiple times, public visibility overrides private visibility, i.e., if an element is imported twice into the same namespace, once using public import and once using private import, it will be public.

public:

private:

protected:

package:

The **Packages** package of **Abstractions** specifies the Package and PackageImport constructs.

4.2.2.27 Packages Diagram

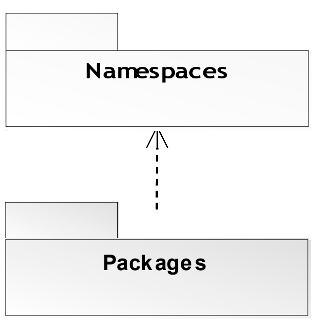


Figure 13 - Packages Diagram

4.2.2.28 Packages

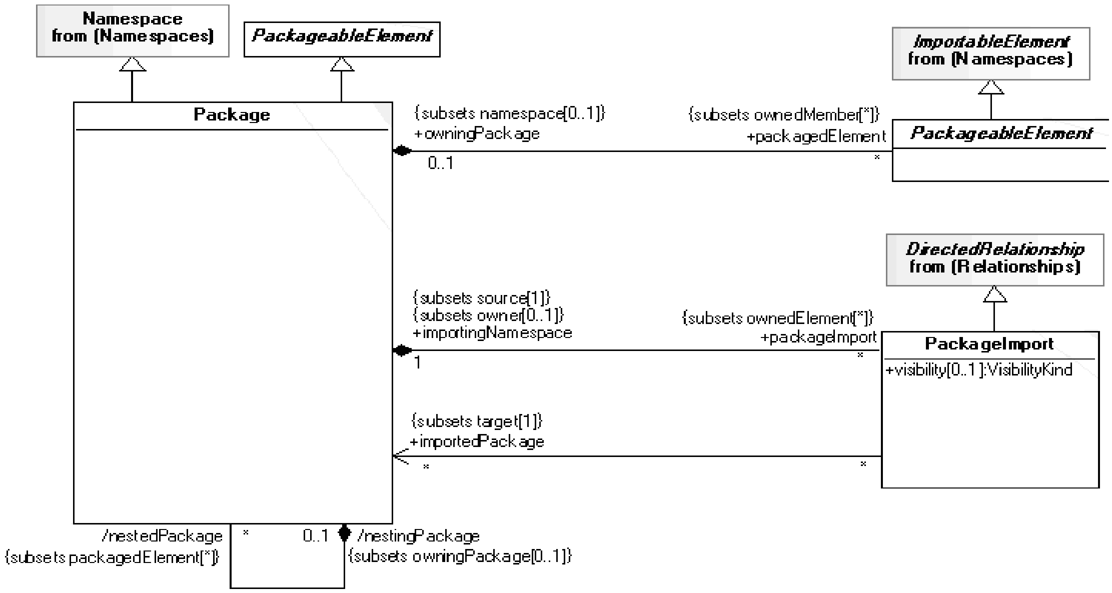


Figure 14 - Packages

4.2.2.29 Package

Package: Packages
isAbstract: No
Generalization: “Namespace” “PackageableElement”

Description

A package is a container for types and other packages. Packages provide a way of grouping types and packages together, which can be useful for understanding and managing a model. A package cannot contain itself.

Associations

packagedElement : PackageableElement [*]	Specifies the packageable elements that are owned by this Package. Subsets *ownedMember*
packageImport : PackageImport [*]	Subsets *ownedElement* Subsets

4.2.2.30 PackageableElement

Package: Packages
isAbstract: Yes
Generalization: “ImportableElement”

Description

A packageable element indicates a named element that may be owned directly by a package.

4.2.2.31 PackageImport

Package: Packages
isAbstract: No
Generalization: “DirectedRelationship”

Description

A package import is a relationship that allows the use of unqualified names to refer to package members from other namespaces.

Description
A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.

Semantics
A package import is a relationship between an importing namespace and a package, indicating that the importing namespace adds the names of the members of the package to its own namespace. Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import.

Attributes

visibility: VisibilityKind [0..1]	Specifies the visibility of the imported PackageableElement within the importing Package. The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import; default value is public.

Associations

importedPackage : Package [*]	Subsets *target*

The TypedElements subpackage of the Abstractions package defines typed elements and their types.

4.2.2.32 TypedElements Package

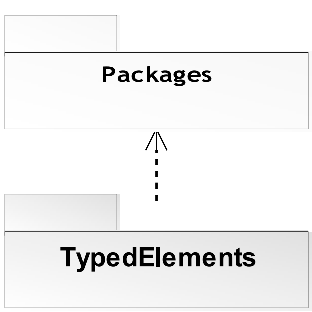


Figure 15 - TypedElements Package

4.2.2.33 Typed Elements

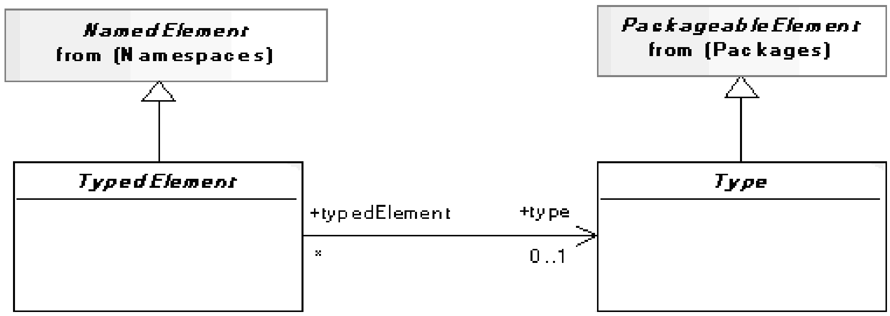


Figure 16 - Typed Elements

4.2.2.34 Type

Package: TypedElements
isAbstract: Yes
Generalization: “PackageableElement”

Description

A **Type** is a **NamedElement** that groups individuals according to some commonality among them, which might be characteristics they can have or constraints they obey. Types can cover any kind of entity, physical or computational, static or dynamic. For example, the type Person groups individual people, like Mary and John. The type declares commonalities among people, for example, they can have names and gender, or obey constraints, such as being genetically related to exactly two other people.

4.2.2.35 TypedElement

Package: TypedElements
isAbstract: Yes
Generalization: “NamedElement”

Description

A typed element is a kind of named element that represents elements with types. Elements with types are instances of TypedElement. A typed element may optionally have no type. The type of a typed element constrains the set of values that the typed element may refer to.

Associations

type : Type [0..1]	The type of the TypedElement.

The Multiplicities subpackage of the Abstractions package defines the metamodel classes used to support the specification of multiplicities for typed elements (such as association ends and attributes), and for specifying whether multivalued elements are ordered or unique.

4.2.2.36 Multiplicities Package

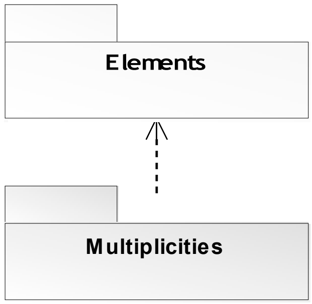


Figure 17 - Multiplicities Package

4.2.2.37 Multiplicities

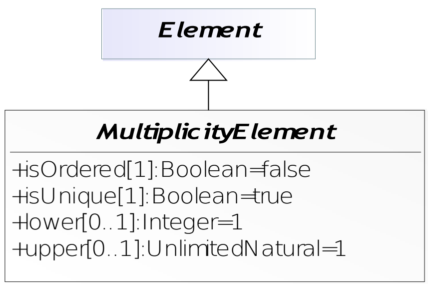


Figure 18 - Multiplicities

4.2.2.38 MultiplicityElement

Package: Multiplicities
isAbstract: Yes
Generalization: "Element"

Description

A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A multiplicity element embeds this information to specify the allowable cardinalities for an instantiation of this element.

Description
A MultiplicityElement is an abstract metaclass that includes optional attributes for defining the bounds of a multiplicity. A MultiplicityElement also includes specifications of whether the values in an instantiation of this element must be unique or ordered.

Semantics
A multiplicity defines a set of integers that define valid cardinalities. Specifically, cardinality C is valid for multiplicity M if M.includesCardinality(C). A multiplicity is specified as an interval of integers starting with the lower bound and ending with the (possibly infinite) upper bound. If a MultiplicityElement specifies a multivalued multiplicity, then an instantiation of this element has a set of values. The multiplicity is a constraint on the number of values that may validly occur in that set. If the MultiplicityElement is specified as ordered (i.e., isOrdered is true), then the set of values in an instantiation of this element is ordered. This ordering implies that there is a mapping from positive integers to the elements of the set of values. If a MultiplicityElement is not multivalued, then the value for isOrdered has no semantic effect. If the MultiplicityElement is specified as unordered (i.e., isOrdered is false), then no assumptions can be made about the order of the values in an instantiation of this element. If the MultiplicityElement is specified as unique (i.e.,

isUnique is true), then the set of values in an instantiation of this element must be unique. If a MultiplicityElement is not multivalued, then the value for isUnique has no semantic effect.

Attributes

isOrdered: Boolean [1]	For a multivalued multiplicity, this attribute specifies whether the values in an instantiation of this element are sequentially ordered. Default is false.
isUnique: Boolean [1]	For a multivalued multiplicity, this attribute specifies whether the values in an instantiation of this element are unique. Default is true.
lower: Integer [0..1]	Specifies the lower bound of the multiplicity interval. Default is one.
upper: UnlimitedNatural [0..1]	Specifies the upper bound of the multiplicity interval. Default is one.

The MultiplicityExpressions subpackage of the Abstractions package extends the multiplicity capabilities to support the use of value expressions for the bounds.

4.2.2.39 MultiplicityExpressions Package

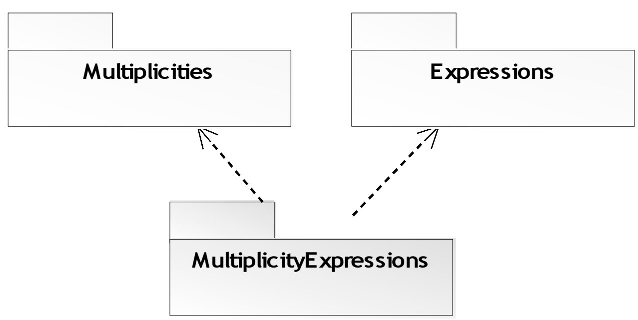


Figure 19 - MultiplicityExpressions Package

4.2.2.40 MultiplicityExpressions

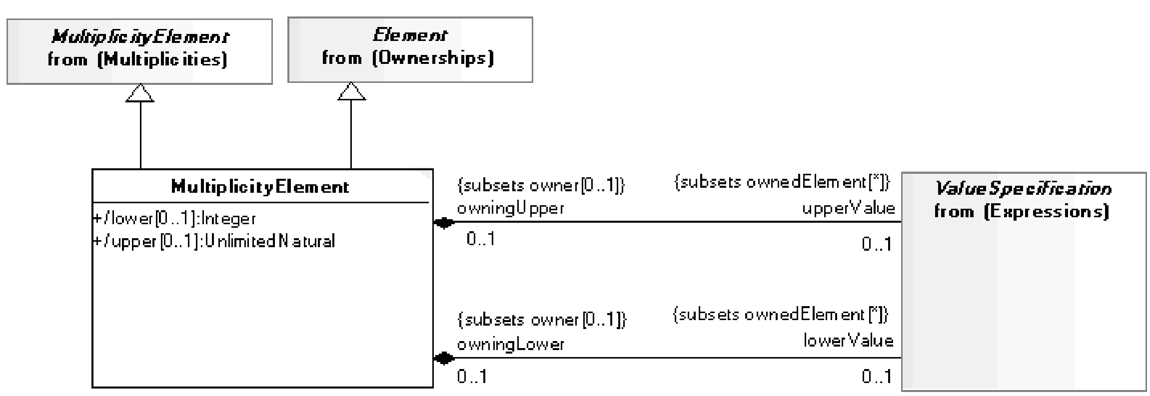


Figure 20 - MultiplicityExpressions

4.2.2.41 MultiplicityElement

Package: MultiplicityExpressions
isAbstract: No
Generalization: "Element" "MultiplicityElement"

Description

MultiplicityElement is specialized to support the use of value specifications to define each bound of the multiplicity.

Attributes

lower: Integer [0..1]	Specifies the lower bound of the multiplicity interval, if it is expressed as an integer. This is a redefinition of the corresponding property from Multiplicities.
upper: UnlimitedNatural [0..1]	Specifies the upper bound of the multiplicity interval, if it is expressed as an unlimited natural. This is a redefinition of the corresponding property from Multiplicities.

Associations

lowerValue : ValueSpecification [0..1]	The specification of the lower bound for this multiplicity. Subsets *ownedElement*
upperValue : ValueSpecification [0..1]	The specification of the upper bound for this multiplicity. Subsets *ownedElement*

The Expressions package in the Abstractions package specifies the general metaclass supporting the specification of values, along with specializations for supporting structured expression trees and opaque, or uninterpreted, expressions. Various UML constructs require or use expressions, which are linguistic formulas that yield values when evaluated in a context.

4.2.2.42 Expressions Package

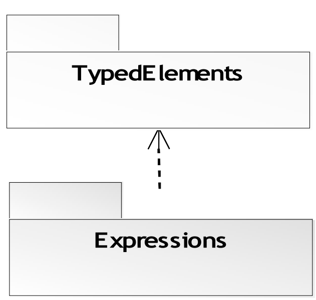


Figure 21 - Expressions Package

4.2.2.43 Expressions

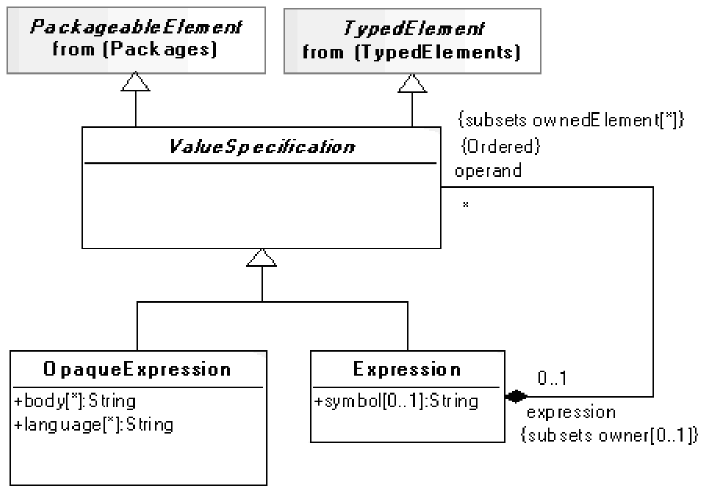


Figure 22 - Expressions

4.2.2.44 Expression

Package: Expressions
isAbstract: No
Generalization: “ValueSpecification”

Description

An expression is a structured tree of symbols that denotes a (possibly empty) set of values when evaluated in a context. An expression represents a node in an expression tree, which may be non-terminal or terminal. It defines a symbol, and has a possibly empty sequence of operands that are value specifications.

Attributes

symbol: String [0..1]	The symbol associated with the node in the expression tree.

Associations

operand : ValueSpecification [*]	Specifies a sequence of operands. Subsets *ownedElement*

4.2.2.45 OpaqueExpression

Package: Expressions
isAbstract: No
Generalization: “ValueSpecification”

Description

An opaque expression is an uninterpreted textual statement that denotes a (possibly empty) set of values when evaluated in a context.

Description
An opaque expression contains language-specific text strings used to describe a value or values, and an optional specification of the languages. One predefined language for specifying expressions is OCL. Natural language or programming languages may also be used.

Attributes

body: String [*]	The text of the expression, possibly in multiple languages.
language: String [*]	Specifies the languages in which the expression is stated. The interpretation of the expression body depends on the language. If languages are unspecified, it might be implicit from the expression body or the context. Languages are matched to body strings by order.

4.2.2.46 ValueSpecification

Package: Expressions
isAbstract: Yes
Generalization: "PackageableElement" "TypedElement"

Description

A value specification is the specification of a (possibly empty) set of instances, including both objects and data values.

Description
ValueSpecification is an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated.

The Literals package in the Abstractions package specifies metaclasses for specifying literal values.

4.2.2.47 Literals Package

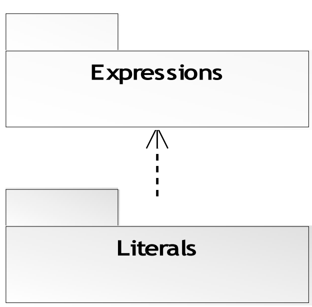


Figure 23 - Literals Package

4.2.2.48 Literals

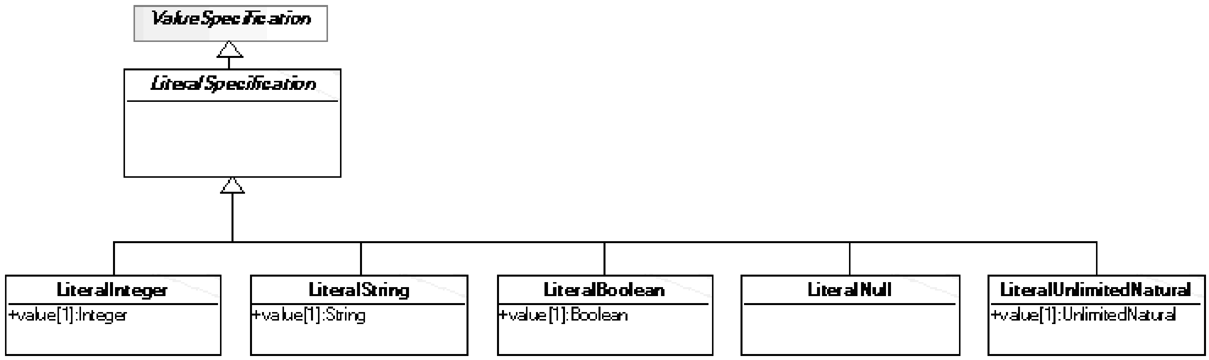


Figure 24 - Literals

4.2.2.49 LiteralBoolean

Package: Literals
isAbstract: No
Generalization: "LiteralSpecification"

Description

A literal Boolean is a specification of a Boolean value.

Description
A literal Boolean contains a Boolean-valued attribute.

Semantics
A LiteralBoolean specifies a constant Boolean value.

Notation
A LiteralBoolean is shown as either the word "true" or the word "false," corresponding to its value.

Attributes

value: Boolean [1]

4.2.2.50 LiteralInteger

Package: Literals
isAbstract: No
Generalization: "LiteralSpecification"

Description

A literal integer is a specification of an integer value.

Description
A literal integer contains an Integer-valued attribute.

Semantics
A LiteralInteger specifies a constant Integer value.

Notation
A LiteralInteger is typically shown as a sequence of digits.

Attributes

value: Integer [1]

4.2.2.51 LiteralNull

Package: Literals
isAbstract: No
Generalization: "LiteralSpecification"

Description

A literal null specifies the lack of a value.

Description
A literal null is used to represent null (i.e., the absence of a value).

Semantics
LiteralNull is intended to be used to explicitly model the lack of a value.

Notation
Notation for LiteralNull varies depending on where it is used. It often appears as the word "null." Other notations are described for specific uses.

4.2.2.52 LiteralSpecification

Package: Literals
isAbstract: Yes
Generalization: "ValueSpecification"

Description

A literal specification identifies a literal constant being modeled.

Description
A literal specification is an abstract specialization of ValueSpecification that identifies a literal constant being modeled.

4.2.2.53 LiteralString

Package: Literals
isAbstract: No
Generalization: "LiteralSpecification"

Description

A literal string is a specification of a string value.

Description
A literal string contains a String-valued attribute.

Semantics
A LiteralString specifies a constant String value.

Notation
A LiteralString is shown as a sequence of characters within double quotes. The character set used is unspecified.

Attributes

value: String [1]

4.2.2.54 LiteralUnlimitedNatural

Package: Literals
isAbstract: No
Generalization: “LiteralSpecification”

Description

A literal unlimited natural is a specification of an unlimited natural number.

Description
A literal unlimited natural contains an UnlimitedNatural-valued attribute.

Semantics
A LiteralUnlimitedNatural specifies a constant UnlimitedNatural value.

Notation
A LiteralUnlimitedNatural is shown either as a sequence of digits or as an asterisk (*), where the asterisk denotes unlimited (and not infinity).

Attributes

value: UnlimitedNatural [1]

The Constraints subpackage of the Abstractions package specifies the basic building blocks that can be used to add additional semantic information to an element.

4.2.2.55 Constraints Package

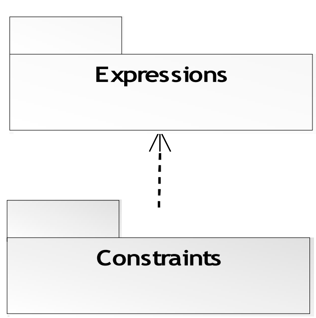


Figure 25 - Constraints Package

4.2.2.56 Constraints

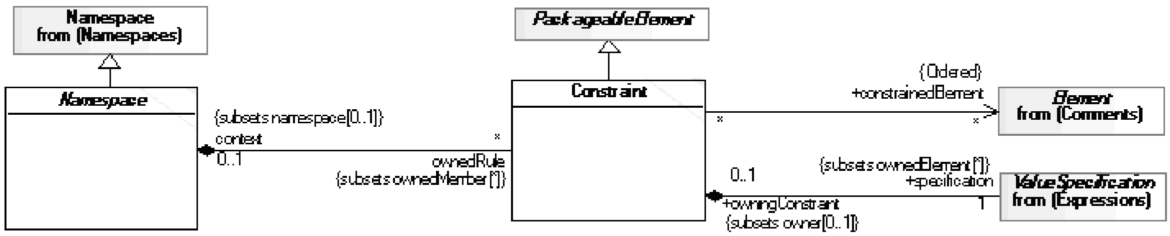


Figure 26 - Constraints

4.2.2.57 Constraint

Package: Constraints
isAbstract: No
Generalization: "PackageableElement"

Description

A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.

Description

Constraint contains a ValueSpecification that specifies additional semantics for one or more elements. Certain kinds of constraints (such as an association "xor" constraint) are predefined in UML, others may be user-defined. A user-defined Constraint is described using a specified language, whose syntax and interpretation is a tool responsibility. One predefined language for writing constraints is OCL. In some situations, a programming language such as Java may be appropriate for expressing a constraint. In other situations natural language may be used. Constraint is a condition (a Boolean expression) that restricts the extension of the associated element beyond what is imposed by the other language constructs applied to the element. Constraint contains an optional name, although they are commonly unnamed.

Semantics

A Constraint represents additional semantic information attached to the constrained elements. A constraint is an assertion that indicates a restriction that must be satisfied by a correct design of the system. The constrained elements are those elements required to evaluate the constraint specification. In addition, the context of the Constraint may be accessed, and may be used as the namespace for interpreting names used in the specification. For example, in OCL "self" is used to refer to the context element. Constraints are often expressed as a text string in some language. If a formal language such as OCL is used, then tools may be able to verify some aspects of the constraints. In general there are many possible kinds of owners for a Constraint. The only restriction is that the owning element must have access to the constrainedElements. The owner of the Constraint will determine when the constraint specification is evaluated. For example, this allows an Operation to specify if a Constraint represents a precondition or a postcondition.

Associations

constrainedElement : Element [*]	The ordered set of Elements referenced by this Constraint.
specification : ValueSpecification [1]	A condition that must be true when evaluated in order for the constraint to be satisfied. Subsets *ownedElement*

4.2.2.58 Namespace

Package: Constraints
isAbstract: Yes
Generalization: "Namespace"

Description

A namespace is a named element that can own other named elements. Each named element may be owned by at most one namespace. A namespace provides a means for identifying named elements by name. Named elements can be identified by name in a namespace either by being directly owned by the namespace or by being introduced into the namespace by other means (e.g., importing or inheriting). Namespace is an abstract metaclass.

Associations

ownedRule : Constraint [*]	Specifies a set of Constraints owned by this Namespace. Subsets *ownedMember*

The Classifiers package in the Abstractions package specifies an abstract generalization for the classification of instances according to their features.

4.2.2.59 Classifiers Package

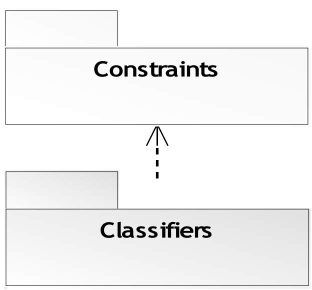


Figure 27 - Classifiers Package

4.2.2.60 Classifiers

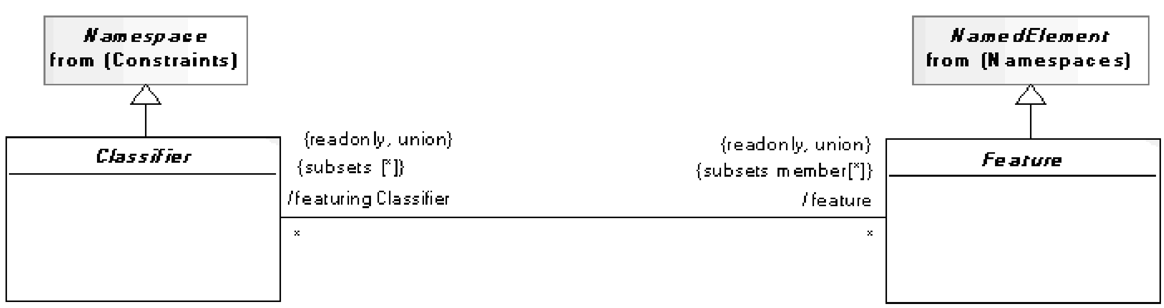


Figure 28 - Classifiers

4.2.2.61 Classifier

Package: Classifiers
isAbstract: Yes
Generalization: "Namespace"

Description

A classifier is a classification of instances - it describes a set of instances that have features in common.

Description
A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

Associations

feature : Feature [*]	Specifies each feature defined in the classifier. This is a derived union. Subsets *member*

4.2.2.62 Feature

Package: Classifiers
isAbstract: Yes
Generalization: "NamedElement"

Description

Description
A feature declares a behavioral or structural characteristic of instances of classifiers. Feature is an abstract metaclass.

Semantics
A Feature represents some characteristic for its featuring classifiers. A Feature can be a feature of multiple classifiers.

Associations

featuringClassifier : Classifier [*]	The Classifiers that have this Feature as a feature. This is a derived union. Subsets

The Super package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.

4.2.2.63 Super Package

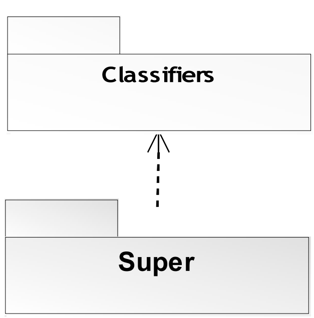


Figure 29 - Super Package

4.2.2.64 Super

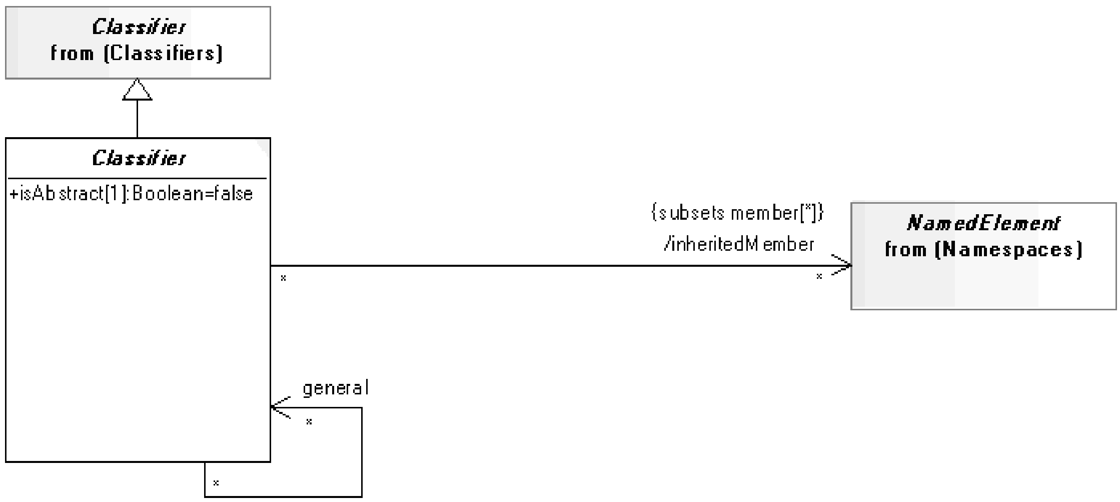


Figure 30 - Super

4.2.2.65 Classifier

Package: Super
isAbstract: Yes
Generalization: “Classifier”

Description

A classifier can specify a generalization hierarchy by referencing its general classifiers.

Attributes

isAbstract: Boolean [1]	If true, the Classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization relationships). Default value is false.

Associations

inheritedMember : NamedElement [*]	Specifies all elements inherited by this classifier from the general classifiers. This is a derived association. Subsets *member*

The Generalizations package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.

4.2.2.66 Generalizations Package

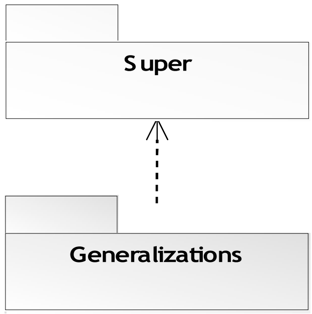


Figure 31 - Generalizations Package

4.2.2.67 Generalizations

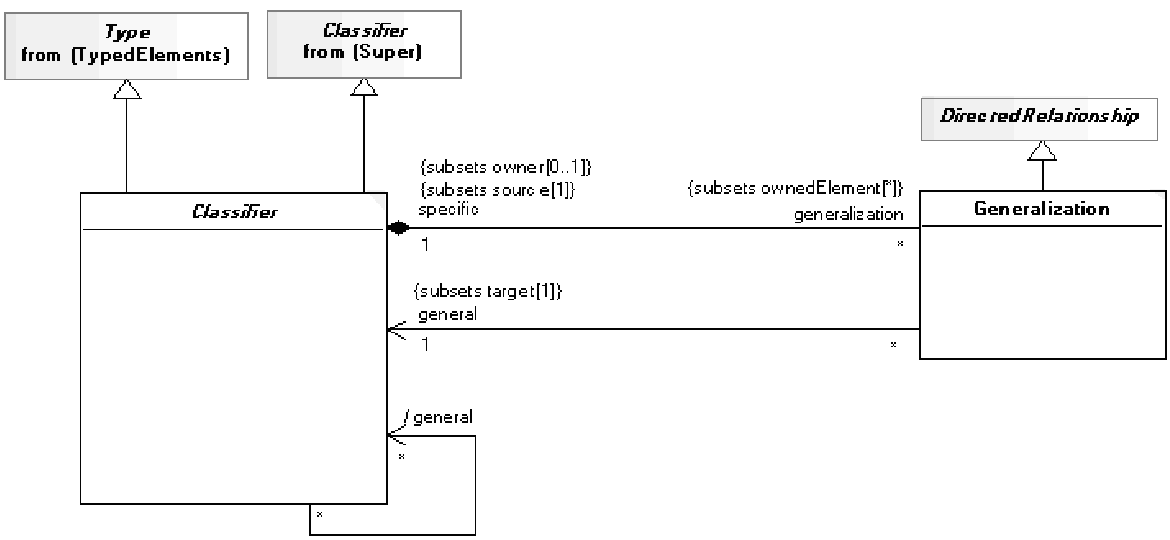


Figure 32 - Generalizations

4.2.2.68 Generalization

Package: Generalizations
isAbstract: No

Generalization: “DirectedRelationship”

Description

A generalization between two types means each instance of the specific type is also an instance of the general type. Any specification applying to instances of the general type also apply to instances of the specific type.

Associations

general : Classifier [1]	References the general classifier in the Generalization relationship. Subsets *target*

4.2.2.69 Classifier

Package: Generalizations
isAbstract: Yes
Generalization: “Classifier” “Type”

Description

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers.

Semantics
A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an (indirect) instance of the general Classifier. The specific semantics of how generalization affects each concrete subtype of Classifier varies. A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.

Associations

generalization : Generalization [*]	generalization specifies the more general super-type of the type Subsets *ownedElement* Subsets

The StructuralFeatures package of the Abstractions package specifies an abstract generalization of structural features of classifiers.

4.2.2.70 Structural Features Package

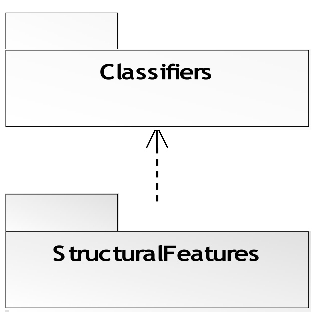


Figure 33 - Structural Features Package

4.2.2.71 Structural Features

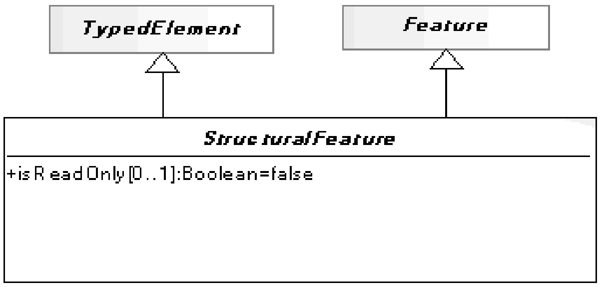


Figure 34 - Structural Features

4.2.2.72 StructuralFeature

Package: StructuralFeatures
isAbstract: Yes
Generalization: “Feature” “TypedElement”

Description

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier.

Description
A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier. Structural feature is an abstract metaclass.

Semantics
A structural feature specifies that instances of the featuring classifier have a slot whose value or values are of a specified type.

Attributes

isReadOnly: Boolean [0..1]	States whether the feature’s value may be modified by a client. Default is false.

The BehavioralFeatures subpackage of the Abstractions package specifies the basic classes for modeling dynamic features of model elements.

4.2.2.73 Behavioral Features Package

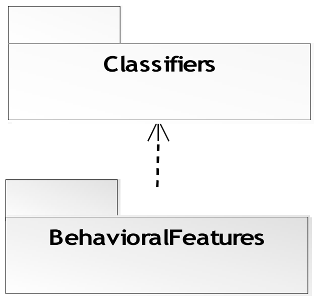


Figure 35 - Behavioral Features Package

4.2.2.74 Behavioral Features

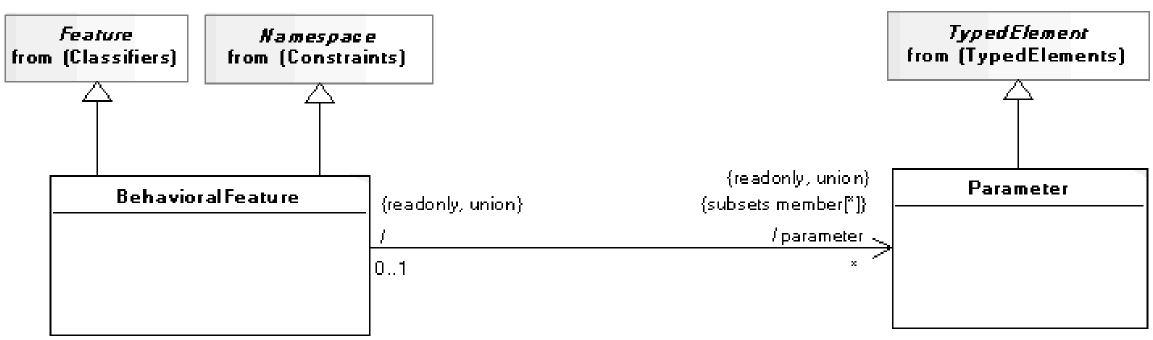


Figure 36 - Behavioral Features

4.2.2.75 BehavioralFeature

Package: BehavioralFeatures
isAbstract: No
Generalization: “Feature” “Namespace”

Description

A behavioral feature is a feature of a classifier that specifies an aspect of the behavior of its instances.

Description
A behavioral feature is a feature of a classifier that specifies an aspect of the behavior of its instances. BehavioralFeature is an abstract metaclass specializing Feature and Namespace. Kinds of behavioral aspects are modeled by subclasses of BehavioralFeature.

Semantics
The list of parameters describes the order and type of arguments that can be given when the BehavioralFeature is invoked.

Associations

parameter : Parameter [*]	Specifies the parameters of the BehavioralFeature. This is a derived union. Subsets *member*

4.2.2.76 Parameter

Package: BehavioralFeatures
isAbstract: No
Generalization: "TypedElement"

Description

A parameter is a specification of an argument used to pass information into or out of an invocation of a behavioral feature.

Semantics
A parameter specifies arguments that are passed into or out of an invocation of a behavioral element like an operation. A parameter's type restricts what values can be passed. A parameter may be given a name, which then identifies the parameter uniquely within the parameters of the same behavioral feature. If it is unnamed, it is distinguished only by its position in the ordered list of parameters.

The Properties subpackage of the Abstractions package specifies the basic class for modeling structural features of model elements.

4.2.2.77 Properties Package

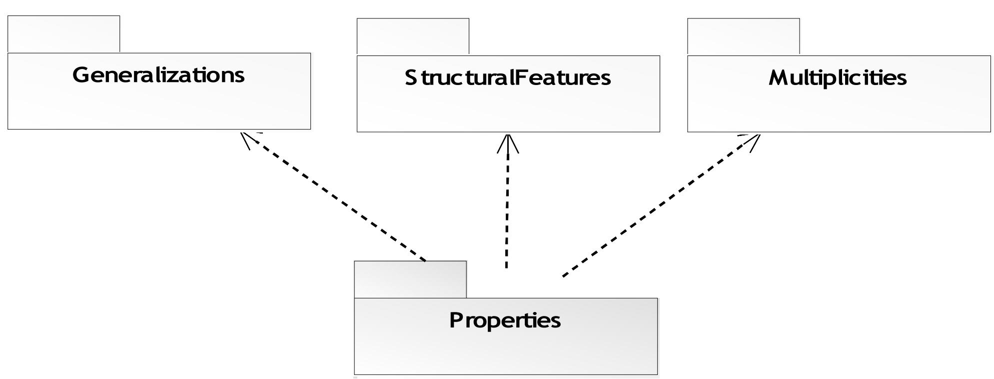


Figure 37 - Properties Package

4.2.2.78 Properties

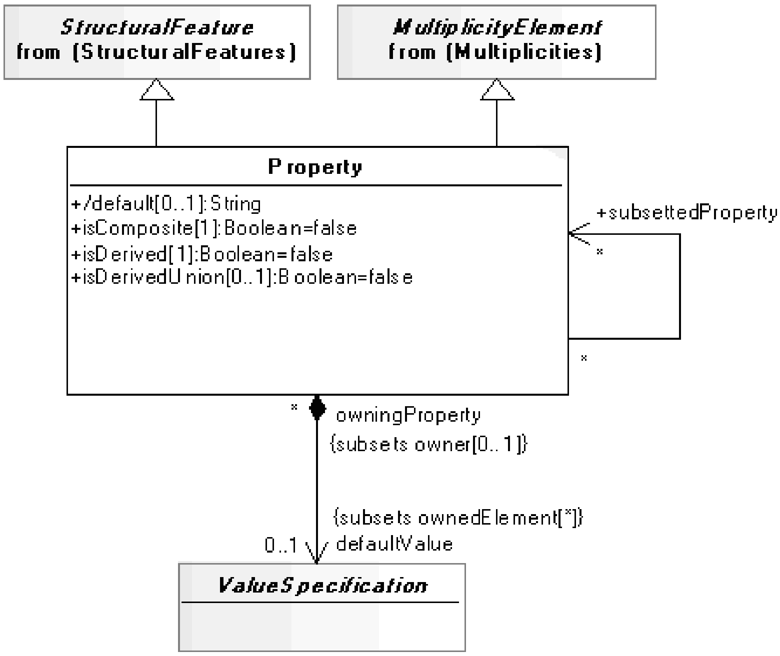


Figure 38 - Properties

4.2.2.79 Property

Package: Properties
isAbstract: No
Generalization: “MultiplicityElement” “StructuralFeature”

Description

A property is a structural feature of a classifier that characterizes instances of the classifier.

Description
Property represents a declared state of one or more instances in terms of a named relationship to a value or values. When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance. Property is indirectly a subclass of TypedElement. The range of valid values represented by the property can be controlled by setting the property's type.

Attributes

default: String [0..1]	A String that is evaluated to give a default value for the Property when an object of the owning Classifier is instantiated.
isComposite: Boolean [1]	This is a derived value, indicating whether the aggregation of the Property is composite or not.
isDerived: Boolean [1]	Specifies whether the Property is derived, i.e., whether its value or values can be computed from other information. The default value is false.
isDerivedUnion: Boolean [0..1]	Specifies whether the property is derived as the union of all of the properties that are constrained to subset it. The default value is false.

Associations

defaultValue : ValueSpecification [0..1]	A ValueSpecification that is evaluated to give a default value for the Property when an object of the owning Classifier is instantiated. Subsets *ownedElement*

The Instances package in the Abstractions package provides for modeling instances of classifiers.

4.2.2.80 Instances Package

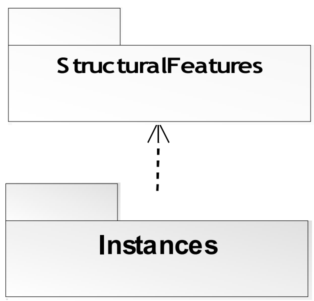


Figure 39 - Instances Package

4.2.2.81 Instances

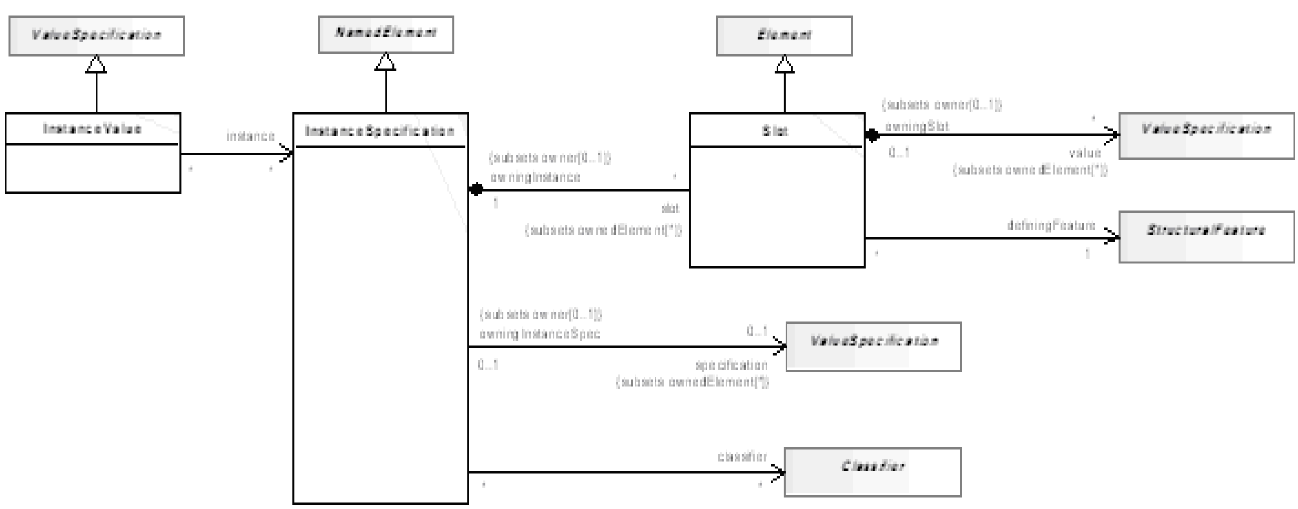


Figure 40 - Instances

4.2.2.82 InstanceSpecification

Package: Instances
isAbstract: No
Generalization: "NamedElement"

Description

An instance specification is a model element that represents an instance in a modeled system.

Description
An instance specification specifies existence of an entity in a modeled system and completely or partially describes the entity. The description includes:

- Classification of the entity by one or more classifiers of which the entity is an instance. If the only classifier specified is abstract, then the instance specification only partially describes the entity.
- The kind of instance, based on its classifier or classifiers. For example, an instance specification whose classifier is a class describes an object of that class, while an instance specification whose classifier is an association describes a link of that association.
- Specification of values of structural features of the entity. Not all structural features of all classifiers of the instance specification need be represented by slots, in which case the instance specification is a partial description.
- Specification of how to compute, derive or construct the instance (optional).

Semantics
An instance specification may specify the existence of an entity in a modeled system. An instance specification may provide an illustration or example of a possible entity in a modeled system. An instance specification describes the entity. These details can be incomplete. The purpose of an instance specification is to show what is of interest about an entity in the modeled system. The entity conforms to the specification of each classifier of the instance specification, and has features with values indicated by each slot of the instance specification. Having no slot in an instance specification for some feature does not mean that the represented entity does not have the feature, but merely that the feature is not of interest in the model. An instance specification can represent an entity at a point in time (a snapshot). Changes to the entity can be modeled using multiple instance specifications, one for each snapshot.

It is important to keep in mind that InstanceSpecification is a model element and should not be confused with the dynamic element that it is modeling. Therefore, one should not expect the dynamic semantics of InstanceSpecification model elements in a model repository to conform to the semantics of the dynamic elements that they represent. When used to provide an illustration or example of an entity in a modeled system, an InstanceSpecification class does not depict a precise run-time structure. Instead, it describes information about such structures. No conclusions can be drawn about the implementation detail of run-time structure. When used to specify the existence of an entity in a modeled system, an instance specification represents part of that system. Instance specifications can be modeled incompletely, required structural features can be omitted, and classifiers of an instance specification can be abstract, even though an actual entity would have a concrete classification.

Associations

classifier : Classifier [*]	The classifier or classifiers of the represented instance. If multiple classifiers are specified, the instance is classified by all of them.
slot : Slot [*]	A slot giving the value or values of a structural feature of the instance. An instance specification can have one slot per structural feature of its classifiers, including inherited features. It is not necessary to model a slot for each structural feature, in which case the instance specification is a partial description. Subsets *ownedElement*
specification : ValueSpecification [0..1]	A specification of how to compute, derive, or construct the instance. Subsets *ownedElement*

4.2.2.83 InstanceValue

Package: Instances
isAbstract: No
Generalization: “ValueSpecification”

Description

An instance value is a value specification that identifies an instance.

Associations

instance : InstanceSpecification [*]	The instance that is the specified value.

4.2.2.84 Slot

Package: Instances
isAbstract: No
Generalization: “Element”

Description

A slot specifies that an entity modeled by an instance specification has a value or values for a specific structural feature.

Description
A slot is owned by an instance specification. It specifies the value or values for its defining feature, which must be a structural feature of a classifier of the instance specification owning the slot.

Semantics
A slot relates an instance specification, a structural feature, and a value or values. It represents that an entity modeled by the instance specification has a structural feature with the specified value or values. The values in a slot must conform to the defining feature of the slot (in type, multiplicity, etc.).

Associations

definingFeature : StructuralFeature [1]	The structural feature that specifies the values that may be held by the slot.
value : ValueSpecification [*]	The value or values corresponding to the defining feature for the owning instance specification. This is an ordered association. Subsets *ownedElement*

The DataTypes subPackage specifies the DataType, Enumeration, EnumerationLiteral, and PrimitiveType constructs. These constructs are used for defining primitive data types (such as Integer and String) and user-defined enumeration data types. The data types are typically used for declaring the types of the class attributes.

4.2.2.85 Datatypes Package

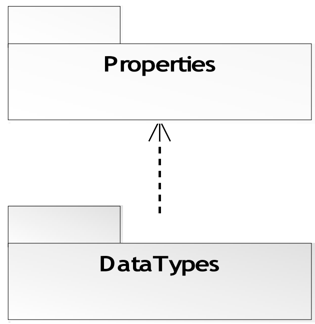


Figure 41 - Datatypes Package

4.2.2.86 Datatypes

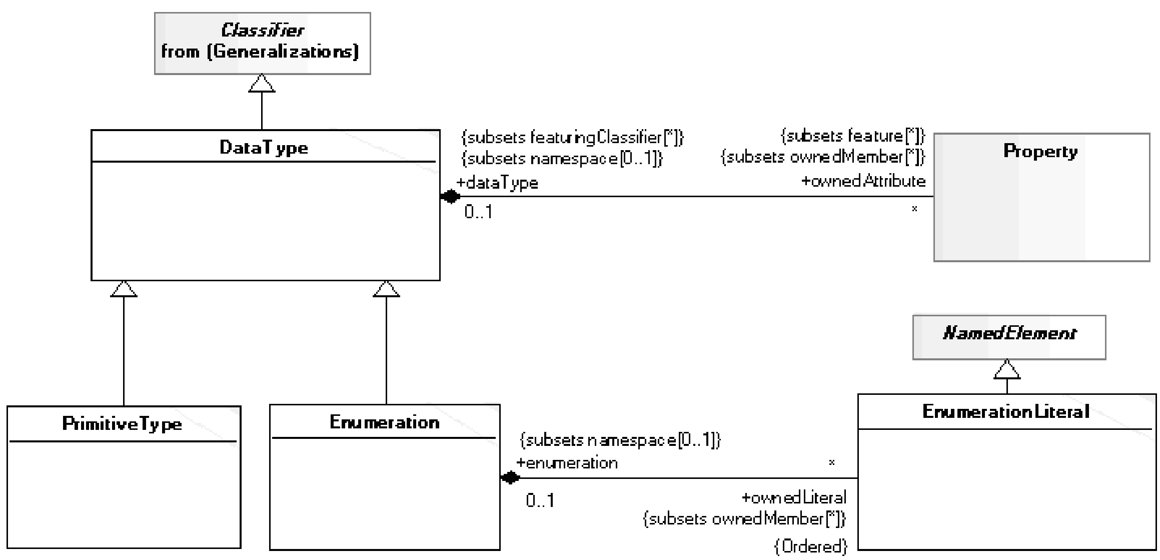


Figure 42 - Datatypes

4.2.2.87 DataType

Package: DataTypes
isAbstract: No
Generalization: "Classifier"

Description

DataType is an abstract class that acts as a common superclass for different kinds of data types. DataType is the abstract class that represents the general notion of being a data type (i.e., a type whose instances are identified only by their value).

Associations

ownedAttribute : Property [*]	The Attributes owned by the DataType. This is an ordered collection. Subsets *feature* Subsets *ownedMember*

4.2.2.88 Enumeration

Package: DataTypes
isAbstract: No
Generalization: "DataType"

Description

An enumeration defines a set of literals that can be used as its values.
An enumeration defines a finite ordered set of values, such as {red, green, blue}. The values denoted by typed elements whose type is an enumeration must be taken from this set.

Associations

ownedLiteral : EnumerationLiteral [*]	The ordered set of literals for this Enumeration. Subsets *ownedMember*

4.2.2.89 EnumerationLiteral

Package: DataTypes
isAbstract: No
Generalization: "NamedElement"

Description

An enumeration literal is a value of an enumeration.

4.2.2.90 PrimitiveType

Package: DataTypes
isAbstract: No
Generalization: "DataType"

Description

A primitive type is a data type implemented by the underlying infrastructure and made available for modeling.

4.2.2.91 Redefinitions Package

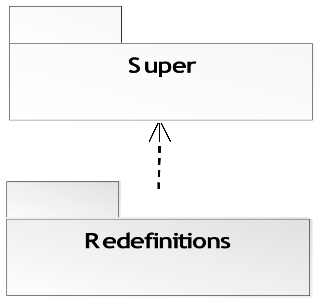


Figure 43 - Redefinitions Package

4.2.2.92 Redefinitions

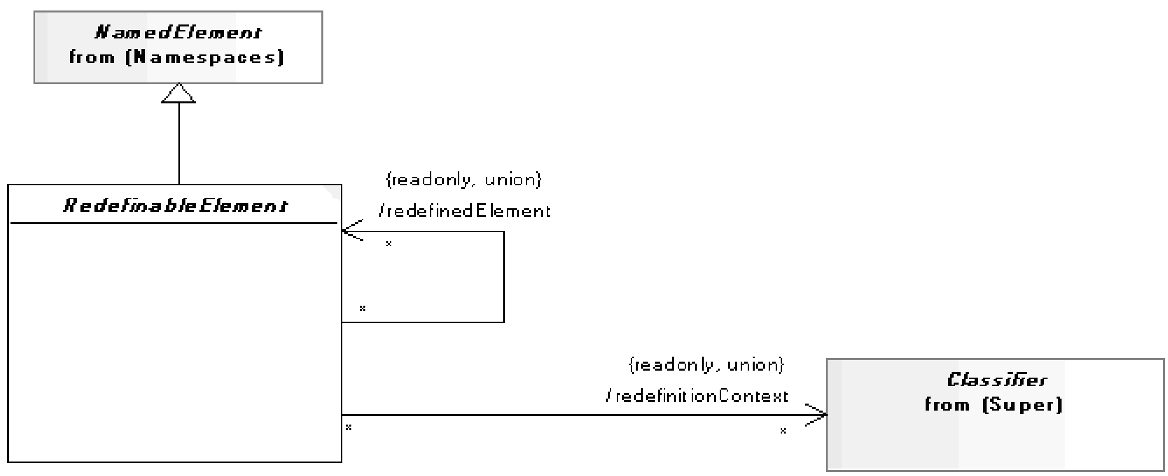


Figure 44 - Redefinitions

4.2.2.93 RedefinableElement

Package: Redefinitions
isAbstract: Yes
Generalization: “NamedElement”

Description

A redefinable element is an element that, when defined in the context of a classifier, can be redefined more specifically or differently in the context of another classifier that specializes (directly or indirectly) the context classifier.

Description
A redefinable element is a named element that can be redefined in the context of a generalization. RedefinableElement is an abstract metaclass.

Semantics

A RedefinableElement represents the general ability to be redefined in the context of a generalization relationship. The detailed semantics of redefinition varies for each specialization of RedefinableElement. A redefinable element is a specification concerning instances of a classifier that is one of the element's redefinition contexts. For a classifier that specializes that more general classifier (directly or indirectly), another element can redefine the element from the general classifier in order to augment, constrain, or override the specification as it applies more specifically to instances of the specializing classifier. A redefining element must be consistent with the element it redefines, but it can add specific constraints or other details that are particular to instances of the specializing redefinition context that do not contradict invariant constraints in the general context. A redefinable element may be redefined multiple times. Furthermore, one redefining element may redefine multiple inherited redefinable elements.

Semantic Variation Points

There are various degrees of compatibility between the redefined element and the redefining element, such as name compatibility (the redefining element has the same name as the redefined element), structural compatibility (the client visible properties of the redefined element are also properties of the redefining element), or behavioral compatibility (the redefining element is substitutable for the redefined element). Any kind of compatibility involves a constraint on redefinitions. The particular constraint chosen is a semantic variation point.

Associations

redefinitionContext : Classifier [*]	References the contexts that this element may be redefined from. This is a derived union.

4.3 Condition Model

4.3.1 Introduction

The Condition Model is for specifying boolean expressions that constrain model elements or capture statements. It defines specialized conditions that are represented as free text, as expressions with particular results, and as boolean combinations of other conditions.

Conditions are boolean ValueSpecifications that constrain some element in the models. They are true if their descriptions hold in the current state of the world, possibly including executions, and false otherwise. Opaque Conditions are Conditions that are expressed in free text. Fact Conditions are Conditions that are true when the two value specifications to which they refer yield equal values, and false otherwise. Compound Conditions are Conditions that provide for combining other conditions with Boolean operators, such as "and" and "or." Statements are boolean ValueSpecifications that do not constrain anything. They are used to integrate with rule models.

4.3.2 Metamodel

The Condition Model is for specifying boolean expressions that constrain model elements or capture statements. It defines specialized conditions that are represented as free text, as expressions with particular results, and as boolean combinations of other conditions.

4.3.2.1 Condition Model Diagram

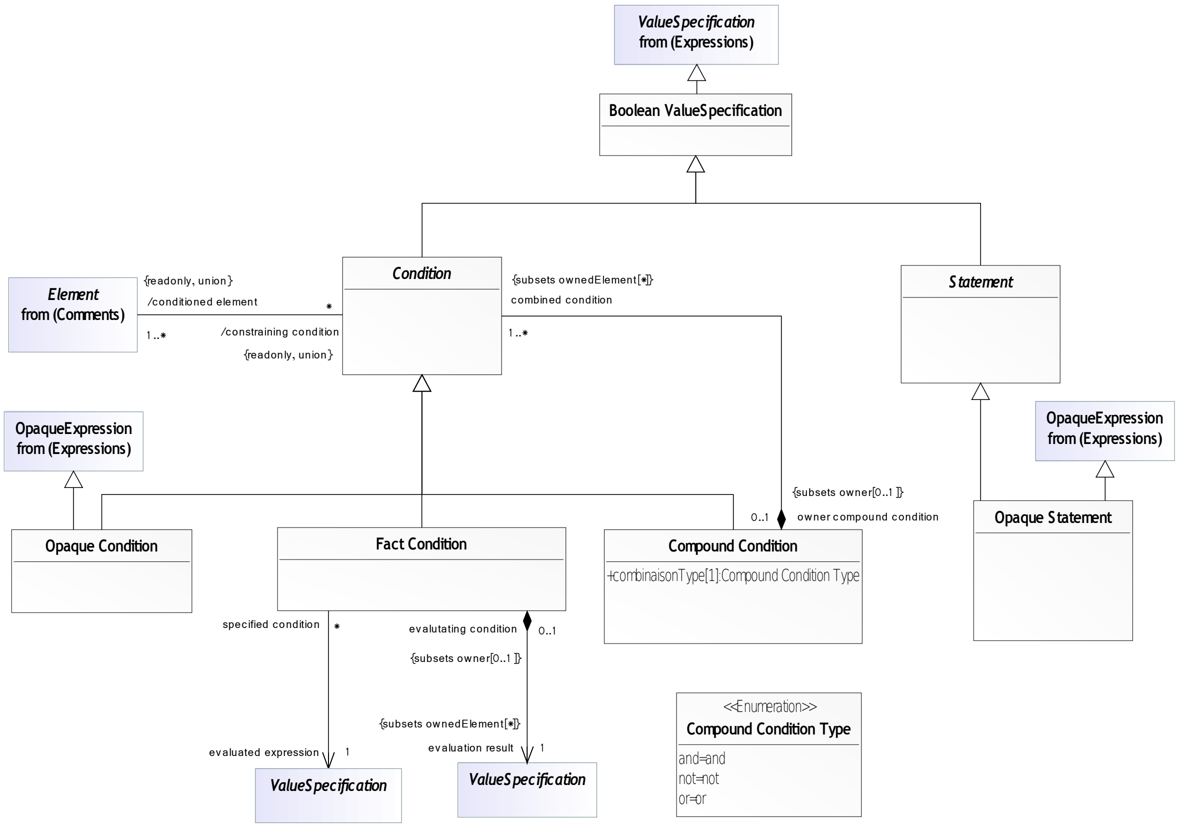


Figure 45 - Condition Model Diagram

4.3.2.2 Boolean ValueSpecification

Package: Condition Model
isAbstract: No
Generalization: "ValueSpecification"

Description

Boolean ValueSpecification is a kind of **ValueSpecification** that specifies a boolean value.

Constraint

[1] The **type** of a **Boolean ValueSpecification** must be a boolean.
self.**type** = Boolean

4.3.2.3 Compound Condition

Package: Condition Model
isAbstract: No
Generalization: "Condition"

Description

A **Compound Condition** is a kind of **Condition** that is the combination of other **Conditions**. There are three kinds of **Compound Condition**:

- **or** : the **Compound Condition** is the result of one the **combined condition**
- **and**: the **Compound Condition** is the result of all the **combined condition**
- **not**: the **Compound Condition** is result of the negation of all the **combined condition**.

Attributes

combinaisonType: Compound Condition Type [1]	Boolean operator used to combine conditions.

Associations

combined condition : Condition [1..*]	Condition making up the Compound Condition Subsets *ownedElement*

4.3.2.4 Compound Condition Type

Package: Condition Model
isAbstract: No

Description

Enumeration specifying the different types of **Compound Condition**

and:

not:

or:

4.3.2.5 Condition

Package: Condition Model
isAbstract: Yes
Generalization: "Boolean ValueSpecification"

Description

A **Condition** is a **Boolean ValueSpecification** that constrains some element in the models. Conditions are true if their descriptions hold in the current state of the world, possibly including executions, and false otherwise.

Associations

conditioned element : Element [1..*]	Element being constrained by the Condition. This is a derived union.

4.3.2.6 Fact Condition

Package: Condition Model
isAbstract: No
Generalization: "Condition"

Description

A **Fact Condition** is a **Condition** that is true when the two **ValueSpecification**s to which they refer yield equal values, and false otherwise.

Associations

evaluated expression : ValueSpecification [1]	ValueSpecification evaluated by the Fact Condition.
evaluation result : ValueSpecification [1]	ValueSpecification that represents the result that must be yielded by the evaluation of the evaluated expression for the Fact Condition to be true. Subsets *ownedElement*

4.3.2.7 Opaque Condition

Package: Condition Model
isAbstract: No
Generalization: “Condition” “OpaqueExpression”

Description

An **Opaque Condition** is a **Condition** that can be expressed in free text.

4.3.2.8 Opaque Statement

Package: Condition Model
isAbstract: No
Generalization: “OpaqueExpression” “Statement”

Description

Opaque Statement is a concrete **Statement** that uses **OpaqueExpression** attributes (language and body) to store its expression as a string.

4.3.2.9 Statement

Package: Condition Model
isAbstract: Yes
Generalization: “Boolean ValueSpecification”

Description

Statement is a **Boolean ValueSpecification** that does not constrain anything. **Statement**s are used to integrate with rule models.

4.4 Composition Model

4.4.1 Introduction

The Composition Model is a framework for relating metamodels to the real world entities they ultimately represent, in particular those with interconnected elements in the same organized whole. This facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and execution across business process management suites. The Composition Model enables users and vendors to build libraries of orchestrations and choreographies, including specialization of some orchestrations or choreographies from others. It also enables users and vendors to define their own frameworks for recording data about ongoing orchestrations and choreographies, for example, how long they have been going, who is involved in them, and what resources they are using. The Composition Model provides general capabilities for representing:

1. The interconnection of elements due to their relation to the same other element. For example, the steps in a process are interconnected because they are all parts of the same process. Interconnections can differ depending on this common element. For example, two processes might have the same steps, but in a different order.
2. Interconnections that are composed of other interconnections. For example, the many fine-grained communications between businesses to set up a partnership may be aggregated into a single joint choreography when viewed at a high level.
3. Interconnections between interconnections. For example, when one communication happens before another during a choreography, it is a connection in time between two other connections.
4. User and vendor-defined characteristics of elements, such as cost, person responsible for them, and resources being consumed.

The Composition Model can be applied in many domains, including structural ones, but in BPDM it is applied to modeling of dynamics, specifically to orchestration and choreography. In this domain the elements are steps in orchestrations, or interactions in choreographies, and the interconnections are relationships in time or transfers of information or physical objects between elements. The elements of the Composition Model are specialized in the other BPDM packages for application to these areas.

The first subsection below is the basis for applying BPDM to business process execution and rules, and to understanding the specification in general. The remaining subsections cover the major elements of the Composition Model.

4.4.1.1 Individuals, Models, and Modeling Languages

An individual is any uniquely identifiable thing. For example, it can be an organization, a piece of hardware, or software component, or something more ephemeral like an information object, process, interaction, or event. The only requirement is that it is distinguished from other individuals. Individual processes and interactions occur at particular times, and are variously called performances, enactments, or executions.

A model describes what we would like from individuals (the *model semantics*). For example, a model of a business specifies what is desired from an actual real world business. Some businesses will satisfy these desires, some will not. Individuals that satisfy the model are said to conform to the model. The rules for conformance are the semantics of the model.[1]

A modeling language consists of shorthands for expressing the semantics of a model. Shorthands used in a model can be "expanded" to give the semantics. For example, a common semantic pattern is to say that all individuals of one kind are also of another kind. A shorthand for this is sometimes called "generalization." Generalization might be used in a model to say that businesses are a generalization of small businesses. This is a shorthand for saying any individual that is a small business is also a business.[2]

Individuals exist at the M0 level in OMG's Model Driven Architecture, while models exist at the M1 level, and modeling languages at the M2 level. The term "individual" in this specification refers only to elements that are not in models or

1 The phrase "instance of" is sometimes used to mean the conformance of an individual to a particular model element (which is often called a "class"), but this terminology usually refers to classes as factories for creating instances, rather than classes as categories. For example, if an individual Fido is a Dog, then Fido is also a Mammal, so conforms to both Dog and Mammal, even though normally Fido would not be called an instance of Mammal, because it was not "created" from Mammal.

2 The difference between shorthands and templates is that the expansion of templates are captured in a machine-understandable way, as part of the modeling language. The expansion of shorthands are specified less formally. Shorthands are more susceptible to misinterpretation than templates, leading to communication failures between users and lack of interoperability between tools.

modeling languages, even though the contents of models and modeling languages are uniquely identifiable like any individual. Similarly, the term "model" in this specification refers only to elements that are not individuals or modeling languages, even though a model language may be expressed as a model (metamodel, see below). More examples and explanation are available in Sections 7.9 through 7.12 of the UML Infrastructure, http://doc.omg.org/formal/07-02-06.

A modeling language has two parts:

- The *language syntax* gives the names of the modeling shorthands and how they can be combined. For example, generalization applies between exactly two kinds of things. Syntax alone cannot determine model semantics, because it refers only to model elements, not individuals.[3]

- The *language semantics* specifies how shorthands are expanded into model semantics. For example, generalization in a model expands to individuals of one kind of thing in the model also being individuals of the other. Language semantics builds on syntax, but must refer to individuals to give a syntax its M0 meaning when the syntax is used in a model.

Some syntaxes are better for specifying language semantics than others. In particular, a syntax that identifies model elements categorizing individuals provides a better basis for specifying model semantics. This enables the language semantics to refer to individuals via the model elements that categorize them. BPDM reuses the syntactical element "Classifier" from UML Infrastructure for this purpose.

4.4.1.2 Classifiers

Classifiers group individuals (uniquely identifiable M0 entities, see Individuals, Models and Modeling Languages) according to some commonality among them, which might be characteristics they can have or constraints they obey. Classifiers can cover any kind of entity, physical or computational, static or dynamic. For example, the classifier Person groups individual people, like Mary and John. The classifier declares commonalities among people, for example, they can have names and gender, or obey constraints, such as being genetically related to exactly two other people. The terms "type" is also used to refer to classifiers, as in "John's type is Person."[4]

Classifiers can group individual occurrences of dynamic entities (M0), such as processes and interactions. For example, the classifier Order Process groups individual performances, enactments, or executions of the ordering, where each occurrence happens between particular start and end times. The classifier declares commonalities among the occurrences, for example, that they involve a product or service, or obey constraints, such as having certain steps taken in a certain order.

Generalization is a relationship between Classifiers indicating that M0 individuals of one classifier are also individuals of another classifier. For example, business is a generalization of small business because individual small businesses are also individual businesses. Specialization is the opposite of generalization, for example, small business is a specialization of business. Parts and constraints specified on the general type apply to all individuals conforming to specializations of that type, because those individuals also conform to the more general type. For example, businesses in general attempt to make a profit, so small businesses do also.

4.4.1.3 Composites

Composites are Classifiers specifying the interconnections of individuals that are all related to the same other individual (M0). For example, a company composite specifies the interconnections of departments within each individual company of that type (assuming it is modeled in a value chain manner, rather than just an organization chart). Likewise, an

---

[3] A metamodel specifies syntax by omitting some aspects of the graphical or textual appearance of the language, such as geometric shapes or punctuation. For example, a metamodel might have an element for kinds of things and another for generalization, but no mention of how generalization appears in a graphical or textual language. This is sometimes called "abstract syntax," as distinguished from "concrete syntax," which includes the detailed graphical or textual appearances.

[4] This commonly used terminology is different from the UML Infrastructure, where Types are elements that specify the range of relations (TypedElements), and Classifiers specify the domain of relations (can own typed elements). Classifiers are Types in the Infrastructure, enabling them to specify both the domain and range of typed elements.

orchestration type specifies the sequence of steps in each individual occurrence of that orchestration.

The things interconnected by a composite can have any kind of relation to the composite. They are not necessarily "contained," "owned," or "part of" the composite. For example, choreographies are composites with the communicating businesses entities as "parts," but the businesses entities are not contained by the choreography in any sense.

4.4.1.4 Parts

To clarify the meaning of "Part" in BPDM, it is important to distinguish two senses in ordinary English:

- Part as an individual, for example the Acme Furniture Company with a unique tax identification number.
- Part as a role, as in "part in a play."

These are mutually defining. Parts in the first sense (individuals) play parts in the second sense ("roles"). For example, a person Mary (individual) may play the president (role) in the Acme Furniture Company. Roles map an individual whole into another individual playing that role in the whole. For example, the president role maps Acme Furniture Company to Mary. (The term "role" is used informally in this section. It has a more specialized meaning in other packages of BPDM.)

Typed Parts in BPDM have the second meaning above. Individuals playing a typed part must be of a certain kind (Classifier[5]), and play the part in the context of another type of thing (whole). For example, an individual playing the president part must be a person, and must play the president within an individual company.[6] Individuals playing parts can have any relation to the whole. They are not necessarily "contained," "owned," or "part of" the whole. For example, a person might be modeled as a composite of anatomically contained parts, but still have other typed parts for relations to other people, such as spouses. The typed part spouseOf will have individuals playing that role for other individuals, but the people are not contained within each other. Typed Parts are MultiplicityElements for restricting the number of individuals that play the part. For example, a company might allow no more than five vice-presidents, but require a president, and a choreography might have an interaction that is optional.

Parts in BPDM are a generalization of Typed Parts to include elements in a composite that do not correspond to individuals (M0). For example, process models often have an indicator that some steps happen at the same time. This part of a process model does not correspond to anything identifiable in the M0 occurrences of the process. It just models the constraint that there are suboccurrences happening at the same time. Because of this, these parts do not have a type restriction like Typed Parts do.

Part Groups are Parts that collect together other Parts. Part groups can share parts. The meaning of part groups is given in the specializations of the Composition Model, for example, in the Behavior Model.

4.4.1.5 Part Connections

Connections between typed parts in the composition model specify links between M0 entities playing the typed parts in the same individual (M0). For example, the reporting connection between the president of a company and the CEO means the person playing the president in a particular company will report to the person playing the CEO in the same company. Likewise, the temporal connection between one step and another in a process means that in each occurrence of that process, there is an occurrence of one step that happens after the occurrence of another.

Connections involving untyped parts do not have a predefined meaning in the Composition Model. They are given specialized interpretations in other packages of BPDM, depending on the parts being connected. For example, parts of a process model indicating that some steps happen at the same time are untyped. Connections to and from these parts require special interpretation to reflect this intention.

---

[5] See footnote 50.

[6] Typed parts are equivalent to what are sometimes called "properties" or "attributes." In this terminology, an individual playing a part is called the "value" of the property or attribute. BPDM Typed Parts are a kind of UML Property.

Part Connections can be treated as first-class parts in themselves, by defining classes that are subtypes of both Part Connection and Typed Part, as done in other BPDM packages. This provides connections that have parts, and connections to connections. For example, choreographies are connections between business entities that are composed of many communications between the businesses. These communications are connections also, and occur in a certain order, which are temporal connections between the communications. Choreographies are the type of their M0 performances, enactments, or executions, which are also M0 links between the businesses. Typed connections require the modeler to specify which parts of the type correspond to which parts on the ends of the connection, see the Part Binding subsection below.

Directed Part Connections are Part Connections between two parts that facilitate traversal from one to the other in user (M1) models. Their source and target associations specify the top-level parts (not part paths) that are connected, as typically shown by the arrows in process diagrams. For example, when one step is after another in a process, the arrow between them is modeled as a directed connection, with the earlier step at the source end, and the later step at the target end. Connections in general can connect any number of parts. For example, a business interaction can involve multiple companies.

Conditions may be applied to connections to limit when they apply. For example, one step in a process may happen after another only when certain conditions are true as the process is executing. Irreflexive Conditions are for restricting connections to apply at M0 only between distinct M0 individuals playing the part (or playing the last part in the path). It applies only to connections between typed parts, or paths with at least one typed part.

4.4.1.6 Part Paths

Some connections are between parts of parts. For example, the temporal connections between steps in a process typically indicate that the start of one step is after the end of another, but they might also indicate that the start of one step is after the start of another, or the end of one step is after the end of another, and so on. To distinguish these cases, the parts on each end of the connection must specify which event (start, end) it is referring to “inside” the step on that end.[7] In BPDM individual events at M0 can be identified by parts, and the combination of the step and the event part is a Part Path.

Part Paths enable connections to refer to parts of parts, for example to connect the end and start events in two steps of a process. For generality, it enables connections to refer to parts of parts to any depth. For example, a part path might refer to the time at which the start event in a step occurs, where the time of an event is modeled as a part of the event. This defines a path through three parts.[8] Part Paths can have a short cut to the last element in the path (final target), for convenience. Part Paths and Parts are generalized to Connectable Elements, which are the ends of connections. This enables connections not requiring part paths to refer directly to parts, rather than to part paths with only one element.

4.4.1.7 Derivation and Selection

Derivation is a relationship between Composites that replaces some parts with others. There is no restriction on the number or kinds of parts that can be replaced by a derived composite. Derivation is useful for exploring alternative configurations for a composite. There are no parts or constraints specified on a composite type that are guaranteed to apply to individuals of derived types. A selector specifies the individuals playing a Typed Part. This might be determined by a rule for each M0 whole that contains the part. A special kind of rule is that the individual must be drawn from a set of predetermined individuals.

4.4.2 Metamodel Specification

The Composition Model is a framework for relating metamodels to the real world entities they ultimately represent. It facilitates integration with business process runtimes and rule engines, as well as uniform performance, enactment, and

7 The step must be specified as a part, rather than just the type of thing done at the step, because a process might have more than one step that does the same thing.

8 A path can contain at most one untyped part, which must be at the end of the path, otherwise it would not be possible to navigate through to the end of the path.

execution across business process management suites. The Composition Model enables users and vendors to build libraries of orchestrations and choreographies, including specialization of some orchestrations or choreographies from others. It also enables users and vendors to define their own frameworks for recording data about ongoing orchestrations and choreographies, for example, how long they have been going, who is involved in them, and what resources they are using.

4.4.2.1 Composition Model Diagram

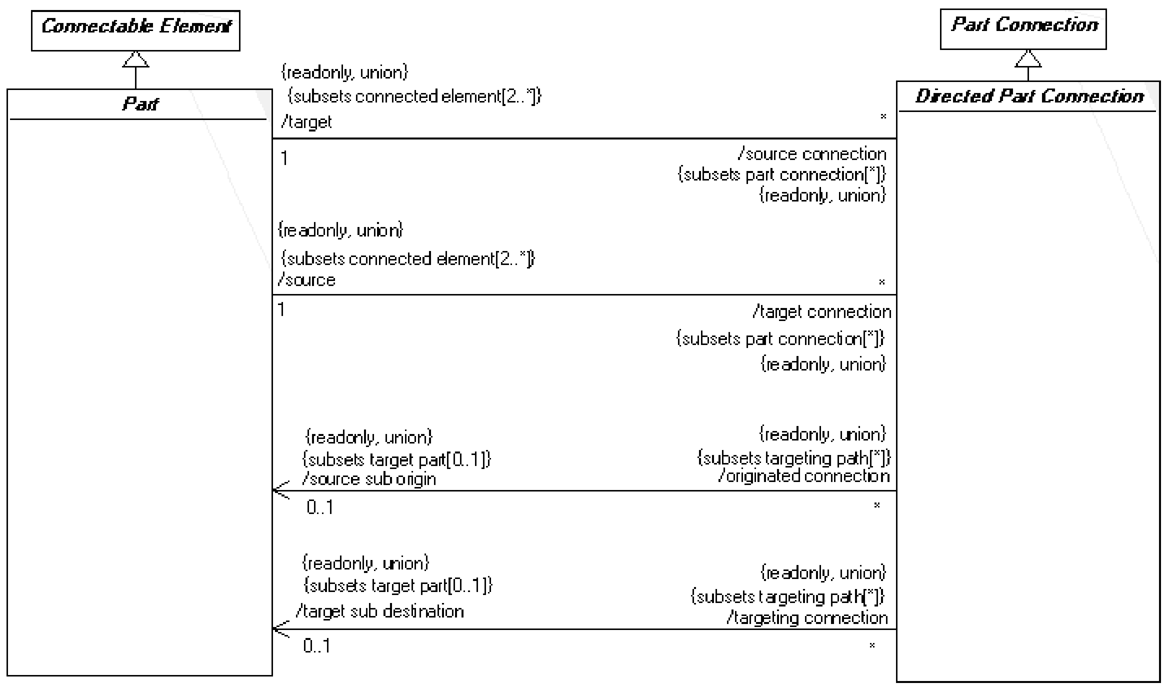


Figure 46 - Composition Model Diagram

4.4.2.2 Directed Part Connection Diagram

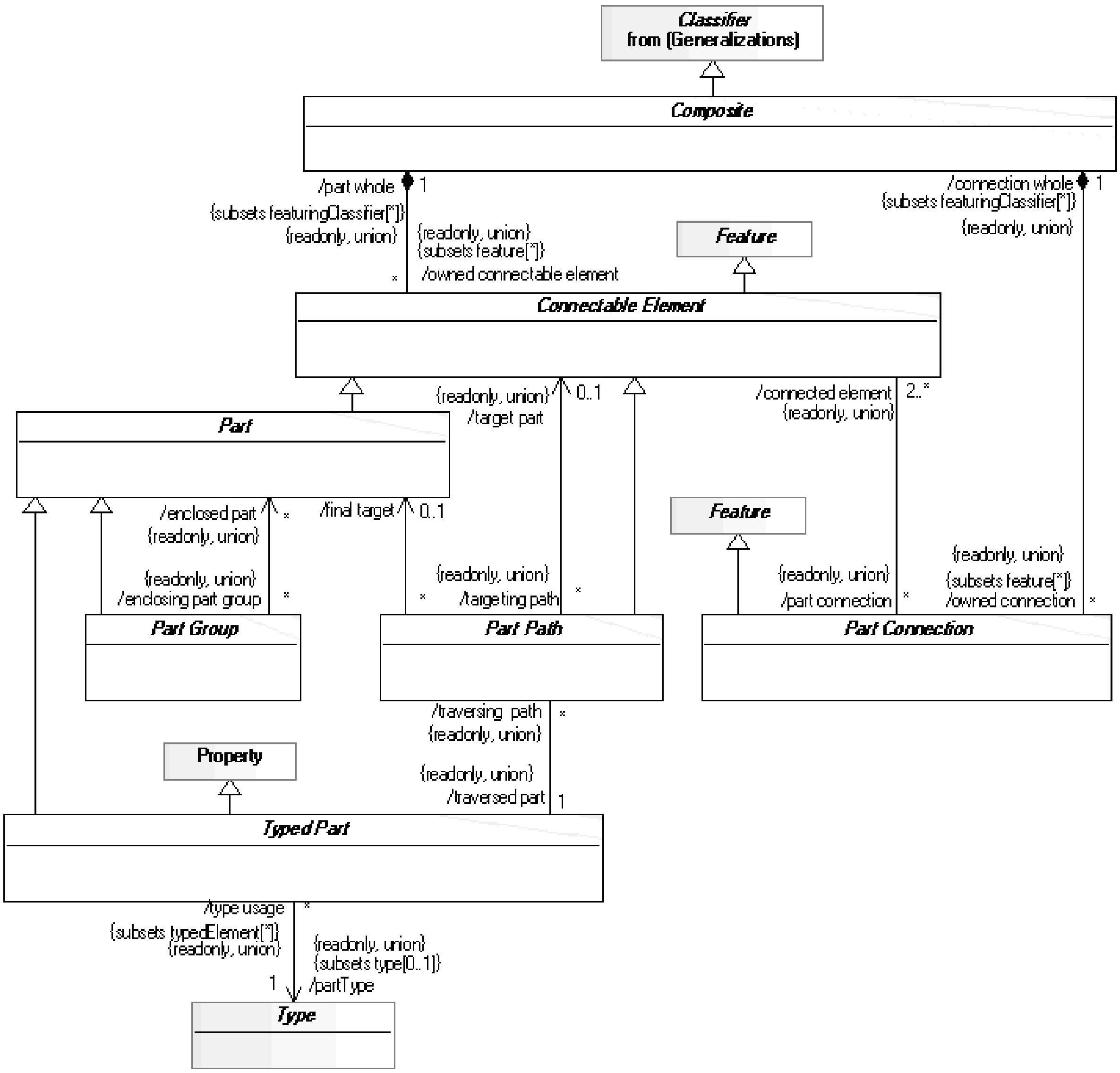


Figure 47 - Directed Part Connection Diagram

4.4.2.3 Part Connection & Condition Diagram

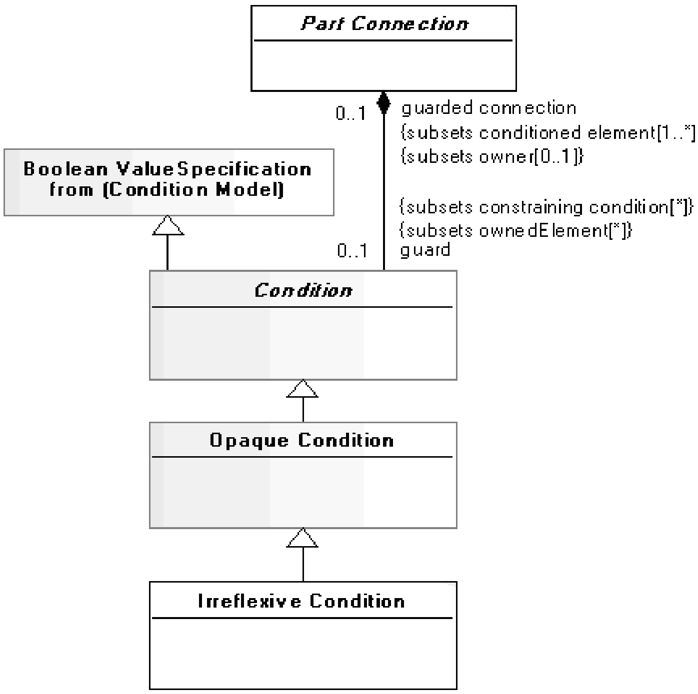


Figure 48 - Part Connection & Condition Diagram

4.4.2.4 Derivation Diagram

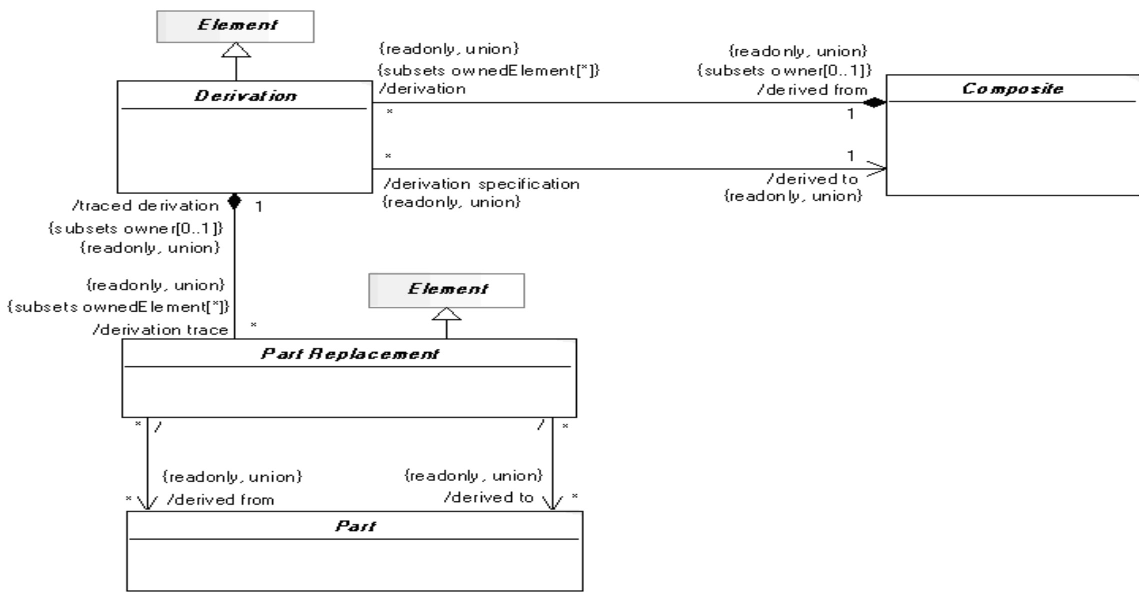


Figure 49 - Derivation Diagram

4.4.2.5 Selection Diagram

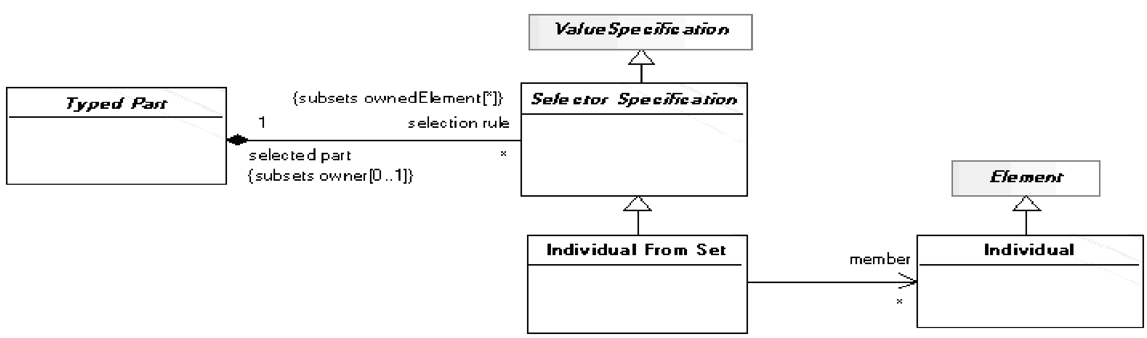


Figure 50 - Selection Diagram

4.4.2.6 Composite

Package: Composition Model
isAbstract: Yes
Generalization: “Classifier”

Description

A **Composite** is a **Classifier** which has an internal structure. It specifies the connections of individuals that are all related to the same other individual (M0). For example, a company type specifies the connections of departments within each individual company of that type (assuming it is modeled in a value chain manner, rather than just an organization chart). Likewise, an orchestration type specifies the sequence of steps in each individual occurrence of that orchestration.

Associations

derivation : Derivation [*]	Derivation that the Composite is a source of This is a derived union. Subsets *ownedElement*
owned connectable element : Connectable Element [*]	Connectable Element owned by the Composite This is a derived union. Subsets *feature*
owned connection : Part Connection [*]	Part Connection owned by the Composite This is a derived union. Subsets *feature*

4.4.2.7 Connectable Element

Package: Composition Model
isAbstract: Yes
Generalization: "Feature"

Description

Connectable Element is the subject of relations between parts through **Part Connection**. **Connectable Element** is a capability shared by **Part** and **Part Path**. Individuals playing parts can have any relation to the whole, they are not necessarily "contained," "owned," or "part of" the whole.

Associations

part connection : Part Connection [*]	Connection connecting the Connectable Element to one or more other Connectable Elements. This is a derived union.

4.4.2.8 Derivation

Package: Composition Model
isAbstract: Yes
Generalization: "Element"

Description

The **Part**s of the **derived to Composite** are the same as the on **derived from Composite**, except for replaced or removed **Part**s, as specified by **derivation trace** , or added parts.

Associations

derivation trace : Part Replacement [*]	Part Replacement owned by the Derivation This is a derived union. Subsets *ownedElement*
derived to : Composite [1]	Derived Composite This is a derived union.

4.4.2.9 Directed Part Connection

Package: Composition Model
isAbstract: Yes

Generalization: “Part Connection”

Description

A **Directed Part Connection** is a kind of **Part Connection** for only two parts, when it is convenient to have standard names referring to the parts on each end (source and target).

Directed Part Connections are designed to facilitate traversal of **Part Connections**. Their source and target associations specify the top-level parts (not **Part Paths**) that are connected, as typically shown by the arrows in process diagrams. For example, when one step is after another in a process, the arrow between them is modeled as a directed connection, with the earlier step at the source part, and the later step at the target part.

Associations

source sub origin : Part [0..1]	This is a derived union. Subsets *target part*
source : Part [1]	Part that is the source of the Directed Part Connection This is a derived union. Subsets *connected element*
target sub destination : Part [0..1]	This is a derived union. Subsets *target part*
target : Part [1]	Part that is the target of the Directed Part Connection This is a derived union. Subsets *connected element*

Constraint

[1] A **Directed Part Connection** must have exactly two **Connectable Elements** (target and source); no more.

4.4.2.10 Individual

Package: Composition Model
isAbstract: No
Generalization: “Element”

Description

Individual instance

4.4.2.11 Individual From Set

Package: Composition Model
isAbstract: No
Generalization: “Selector Specification”

Description

An **Individual From Set** is a kind of **Selector Specification** that provides a list of **Individual** as the potential **Type** of a **Typed Part**.

Associations

member : Individual [*]	Individual member of a Individual From Set selector specification

4.4.2.12 Irreflexive Condition

Package: Composition Model
isAbstract: No
Generalization: "Opaque Condition"

Description

An **Irreflexive Condition** is a kind of **Opaque Condition** that restricts the connection to apply at M0 only to distinct M0 individuals playing the part (or playing the last part in the path). It applies only to connections between **Typed Parts**, or **Part Paths** with at least one **Typed Part**.

4.4.2.13 Part

Package: Composition Model
isAbstract: Yes
Generalization: "Connectable Element"

Description

A **Part** is a **Connectable Element** that is an element of the structure of a **Composite**.

Associations

source connection : Directed Part Connection [*]	Directed Part Connection that the Part is the target of. This is a derived union. Subsets *part connection*
target connection : Directed Part Connection [*]	Directed Part Connection that the part is the source of. This is a derived union. Subsets *part connection*

4.4.2.14 Part Connection

Package: Composition Model
isAbstract: Yes
Generalization: "Feature"

Description

A **Part Connection** is a **Feature** of a composite used to connect its **Connectable Elements**. A **Part Connection** can connect any number of parts. For example, a business interaction can involve multiple companies.

When a **Part Connection** is connecting **Typed Part**, its specifies links between M0 entities playing the typed parts. For example, the reporting connection between the president of a company and the CEO means the person playing the president in a particular company will report to the person playing the CEO in the same company. Likewise, the temporal connection between one step and another in a process means that in each occurrence of that process, there is an occurrence of one step that happens after the occurrence of another.

Conditions may be applied to **Part Connections** to limit when they apply. For example, one step in a process may happen after another only when certain conditions are true as the process is executing.

Associations

connected element : Connectable Element [2..*]	Connectable Element connected by a Part Connection This is a derived union.
guard : Condition [0..1]	Condition evaluated at runtime to determine if the Part Connection is enabled. Subsets *constraining condition* Subsets *ownedElement*

4.4.2.15 Part Group

Package: Composition Model
isAbstract: Yes
Generalization: "Part"

Description

A **Part Group** is a kind of **Connectable Element** that collects other **Connectable Element**s together. A **Part Groups** can share **Connectable Elements**. The meaning of part groups is given in the specializations of the **Composition Model**, for example, in **Behavior Model**.

Associations

enclosed part : Part [*]	Part that is enclosed in a Part Group. A Part can be enclosed in multiple Part Groups This is a derived union.

BPMN Notation

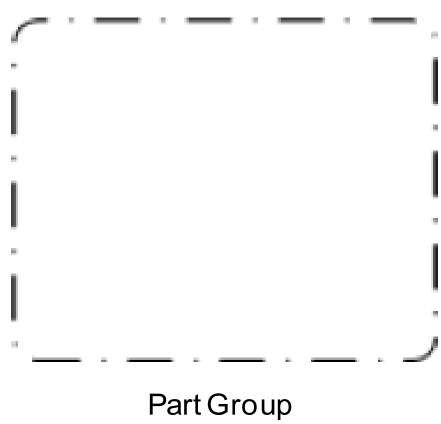


Figure 51 - Part Group Notation

4.4.2.16 Part Path

Package: Composition Model
isAbstract: Yes
Generalization: "Connectable Element"

Description

A **Part Path** connects to a **Part** of a nested **Composite**.
An instance of **Part Path** is introduced for each **traversed part** to a **target part**.

The purpose of **Part Path** is to provide access to parts in a nested composite structure. All models based on the composition model needs to have access to parts within parts, for example:

- Data elements within data elements
- Roles within roles
- Protocols within protocols
- Activities within activities

Part Path and **Part** are generalized to **Connectable Element**, which are the is of **Part Connection**. This enables connections not requiring part paths to refer directly to parts, rather than to part paths with only one element.

Associations

final target : Part [0..1]	leaf Part to which a part path chain is pointing at This is a derived association.
target part : Connectable Element [0..1]	Connectable Element to which the part path is pointing at. This is a derived union.
traversed part : Typed Part [1]	Typed Part being the source of the part path. This part is traversed by the part path in order to reach the target part. This is a derived union.

Constraint

[1] The **target part** must be a **Part** of the **Composite** that owns the **target part**

[1] The **traversed part** must be a **Typed Part** which type is a **Composite**.

4.4.2.17 Part Replacement

Package: Composition Model
isAbstract: Yes
Generalization: "Element"

Description

A **Part Replacement** is used to specify the replacement or removal of **Part**s in **derived to Composite** of a **Derivation**.

Associations

derived from : Part [*]	This is a derived union.
derived to : Part [*]	This is a derived union.

4.4.2.18 Selector Specification

Package: Composition Model
isAbstract: Yes

Generalization: "ValueSpecification"

Description

A **Selector Specification** is a query mechanism used to specify the individuals playing a **Typed Part**.

4.4.2.19 Typed Part

Package: Composition Model
isAbstract: Yes
Generalization: "Part" "Property"

Description

A **Typed Part** is a kind of **Part** that specifies that individuals playing the **Part** in the **Composite** must be of a certain kind (**Type**). For example, an individual playing the president part must be a person, and must play the president within an individual company.

Typed Part is a **Property** for restricting the number of individuals that play the part. For example, a company might allow no more than five vice-presidents, but require a president, and a choreography might have an interaction that is optional.

Associations

partType : Type [1]	Type of the Typed Part This is a derived union. Subsets *type*
selection rule : Selector Specification [*]	Selector Specification used to specify the individual that plays the Typed Part Subsets *ownedElement*
traversing path : Part Path [*]	Part Path that traverses the Typed Part in order to reach a part of its composite type. This is a derived union.

Constraint

[1] The default values for lower and upper (from Abstraction:MultiplicityElement) are 0 and * respectively.

context Typed Part::lower: Integer
init: 0
context Typed Part::upper: UnlimitedInteger
init: *

4.4.2.20 Instance: Irreflexive Condition

Class: Irreflexive Condition

Description

Links

Played End	*Opposite End*
Irreflexive Condition:guard	end/abort

4.5 Course Model

4.5.1 Introduction

The Course Model extends the Composition Model for dynamics. It introduces connections for time ordering of parts (Succession), including time ordering of process lifecycle events, such as starting and ending a process. For example, a succession connects one step in a process to another to indicate that the second step happens sometime after before the first. The same applies to messages in choreography, and to process lifecycle events, for example ,a process always ends sometime after it starts. This facilitates the integration of rule and monitoring systems with models of dynamics, such as orchestration and choreography. The model enables users and vendors to define their own libraries of processes, with their own categorizations and attributes, such as how long a process has been running, and the resources it is using. They can also define their own life cycle events, for example, to define finish statuses and taxonomy of errors.

The Course Model extends the Composition Model with:

- General categories for dynamic entities that extend over time (Happenings Over Time) producing entities that occur at a point in time (Events).
- Dynamic entities that produce lifecycle events, such as starting and ending, enabling the events to be ordered in time (Cousers and Behavioral Events).
- A user (M1) library defining a behavior that produces common behavior lifecycle events, such starting and ending (Behavior Occurrence).
- Conditions for time events and changes in facts.

Happenings are Classifiers for the most general notion of dynamic entities, including processes and events. Happenings at M1 are classifiers for individual M0 happening occurrences, such as individual performances, enactments,or executions of processes, and occurrences of events. Happenings Over Time and Events are Happenings that extend over time, or as occur at a point in time, respectively. Happenings over time produce events, for example, the revenue of a company changes during a business process. A dynamic entity could be either a happening over time or an event, depending on the viewpoint of the application. For example, a package arriving at a business might be treated as a process of signing for it, inspecting it, and routing it to the addressee or it might be treated as simply occurring on a particular day with no additional detail.

Courses are Composites that are also Happenings Over Time. As composites, courses have Happening Parts, which are parts played by happenings. These enable individual M0 courses to be linked to individual M0 happenings, such as individual performances, enactments, or executions of subprocesses and individual M0 lifecycle events. As composites, courses also have Succession connections representing that one part of the course "follows" another in time, and possibly establishes constraints on such followings (Course Parts are introduced just to categorize those Parts that can be related by Successions). Immediate Successions are Successions for specifying that one part of the course immediately follows another, as opposed to following sometime afterwards. Successions have different meanings for typed and un-typed parts:

- For typed course parts, such as Happening Parts, Succession means that an individual dynamic entity playing one typed part will happen at the same time or after another dynamic entity playing another typed part as the course proceeds. These dynamic entities might be steps in a process, interactions in choreography, or events due to these. Immediate Successions are Successions where the dynamic entities being connected happen at the same time. For example, two steps in a process might be required to start at the same time. Typed course parts specify conditions incoming successions must satisfy for dynamic entities playing a part to start, and conditions outgoing successions must satisfy when dynamic entities playing a part come to an end. Predefined conditions requiring all successions to be satisfied (AllSuccession) or only one succession (OneSuccession) are provided in an M1 model library.

- For un-typed course parts, such as Gateways, Successions represent more complex specifications of how dynamic individuals playing typed parts are ordered in time. Parallel Splits are Gateways indicating that the dynamic individuals playing parts following them happen after the dynamic individuals playing the part preceding them. Parallel Joins indicate that the parts (in the sense of individuals) following them happen after the parts preceding them. Exclusive Splits indicate that exactly one of the parts following them will occur after the part preceding them. Exclusive Joins indicate that the part following them will occur after each part that occurs preceding them. Successions with un-typed parts at one or both ends may not have part paths at those ends, including qualification, because there will be no individual playing that part (see Composition Model).

As happenings over time, courses produce Course Events, which are process lifecycle events, such as starting and ending. Event Parts are Happening Parts identifying events for individual M0 courses. For example, an event part for shipping a product can identify the starting event for each individual shipment, such as 8am on a particular day. Event Parts are also Course Parts, enabling them to be connected by Successions. For example, an event part identifying the end of a course succeeds the event part identifying the start. This means the ending of each individual M0 course occurrence, such as an individual shipment, is after the start of that same individual course.

A user (M1) library in the Course Model captures commonly needed aspects of courses as instances of classes in the Course Model. The library defines:

- Course Events representing process lifecycle events, specifically starting and ending of individual courses.
- A taxonomy of M0 happening occurrences rooted at Happening Occurrence, which is a generalization of all M1 dynamic models, including all orchestration and choreography models. All individual (M0) happening occurrences conform to Happening Occurrence, which is the most abstract M1 model of happenings. It generalizes Happening Over Time Occurrences and Event Occurrences, which generalize Course Occurrences and Course Event Occurrences, respectively.
- Event Parts of Course Occurrence for the various Course Events, such as start and end parts. These are typed by the M1 events Start Event and End Event. They can be the source or target for successions, see below.
- Successions between the Event Parts above for M0 time ordering, such as the end of every course being after the start.

Successions can order the event parts of happening parts, such as the start and end parts of packing or shipping in a delivery process. For example, a succession might have the packing part as source and the end part as source event part, while the shipping part is the target, and the start part is the target event part. This means packing must end before shipping starts, specifically, the ending of each individual M0 packing occurrence within a delivery occurrence is before the start of that same individual course. Other combinations of event parts in succession might be one happening part starts after another starts, ends after another ends, or ends after another starts. For convenience, successions that do not specify source or target event parts will have the same effect as successions where these are the end parts and start parts, respectively. Successions do not need to have happening parts as source and target, they can have untyped course parts also, such as gateways.

The library enables users and extenders of BPDM to define their own:

- Parts of courses, for example, a business monitoring model or business runtime model can specialize Course Occurrence to introduce typed parts for the time an individual process starts, how long it has been running, and the resources it is using.
- Taxonomies of courses, for example, a general business process can be specialized for small and large businesses, or business in specific sectors, such as health care or retail. This can be the framework for libraries of reusable business processes.
- Taxonomies of events, for example, to define kinds of errors and introduce error codes.
- Taxonomies of event parts, for example, to take different steps depending on which error ends a course.

The Course Events in the user library are for the starting and ending of courses (Start Event and End Event). Individual (M0) course events play event parts as they occur. The user library (M1) defines event parts for the event types in the library, in particular, individual start events at M0 play start parts, and individual end events at M0 play end parts. Each individual (M0) course occurrence will have exactly one start event and one end event. Inversely, each individual course event must play an event part in exactly one individual course occurrence. For example, an M0 start event plays the start part for exactly one individual course occurrence.

Successions in Course Occurrences inherit to all user-defined course definitions (M1) and all individual (M0) course occurrences (all performances, enactments, and executions). These establish the time order of process lifecycle events, for example, that ending happens after starting. Successions that target parts typed by the Start Event specify a new individual (M0) course. For example, a process definition may indicate that an incoming message creates a new execution of a process by a succession from the message receipt to the start part in the user library. Event parts can be the source or target of Successions, for example, to specify different steps that follow normal and abnormal ends.

Event Conditions are Conditions for specifying that an Event must occur in the context of a particular Happening Over Time for the condition to hold. It generalizes Time events and changes in Facts (also see the Behavior Model). Event Conditions specify that an individual (M0) happening over time must produce a particular kind of event (defined at M1) for the condition to hold. Time Event Condition is specified by referring to a Clock, which is a Happening Over Time that produces Time Events. Time Events have a property for specifying the time in a detailed expression. Fact Change Conditions refer to general propositions becoming true or false due to changes in M0 facts. It is used to integrate with models of rules.

4.5.2 Metamodel Specification

The Course Model extends the Composition Model to connect parts in time (Succession). For example, a succession connects one step in a process to another to indicate that the second step happens after the first. The same applies to messages in choreography.

4.5.2.1 Happening and Event Diagram

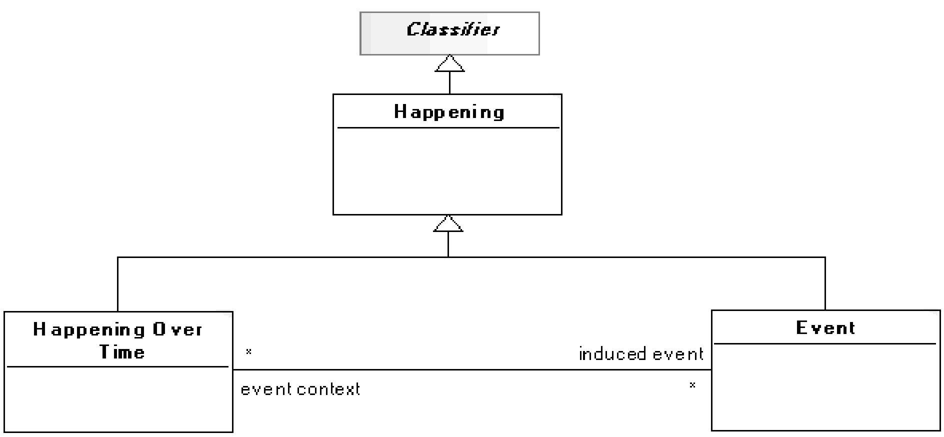


Figure 52 - Happening and Event Diagram

4.5.2.2 Time Event Diagram

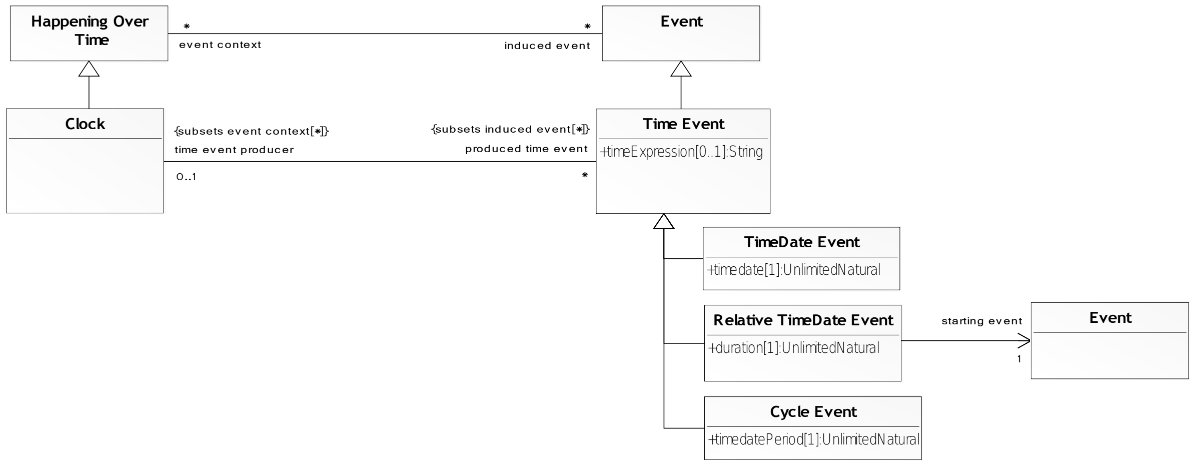


Figure 53 - Time Event Diagram

4.5.2.3 Event Condition Diagram

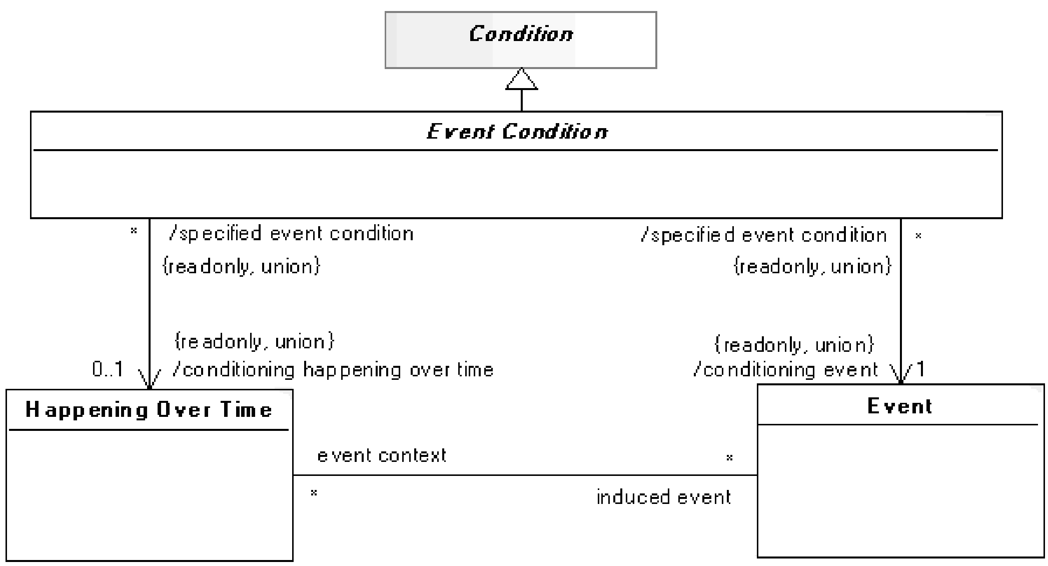


Figure 54 - Event Condition Diagram

4.5.2.4 Time Event Condition Diagram

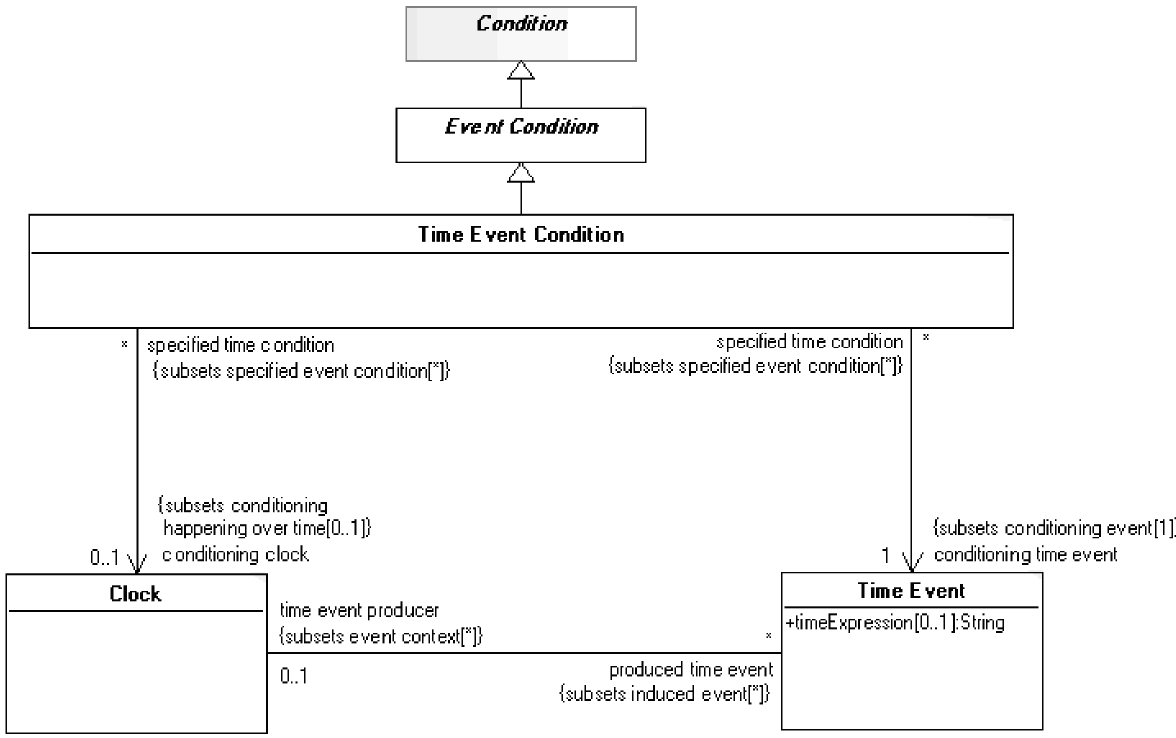


Figure 55 - Time Event Condition Diagram

4.5.2.5 Fact Change Condition Diagram

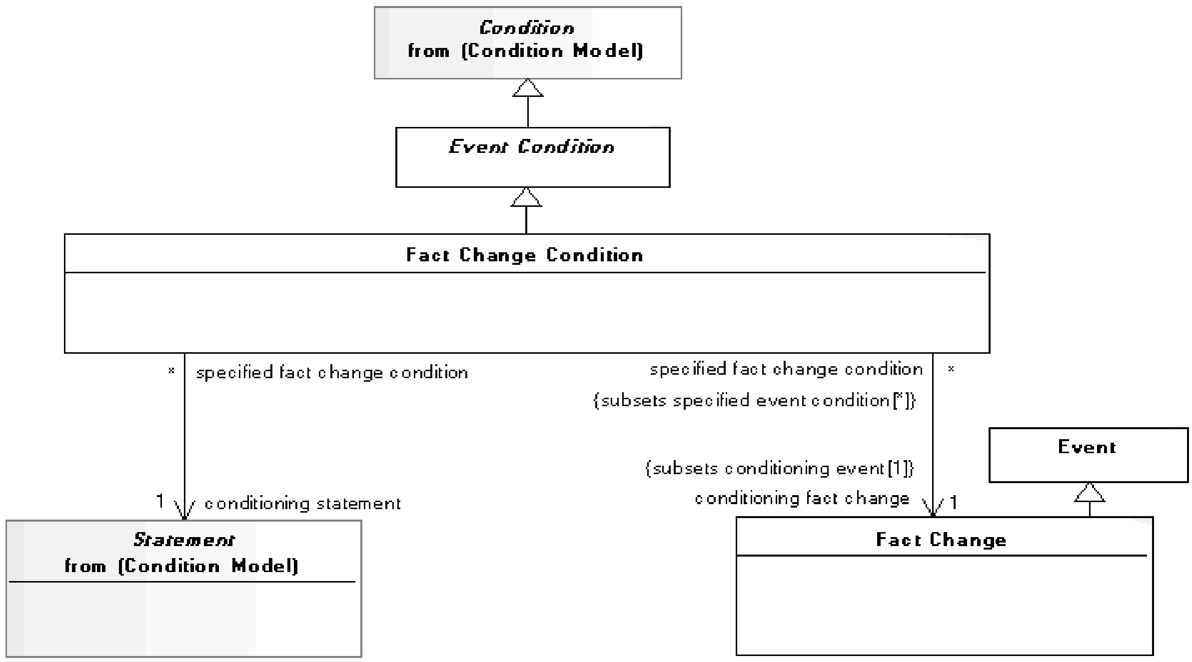


Figure 56 - Fact Change Condition Diagram

4.5.2.6 Course Diagram

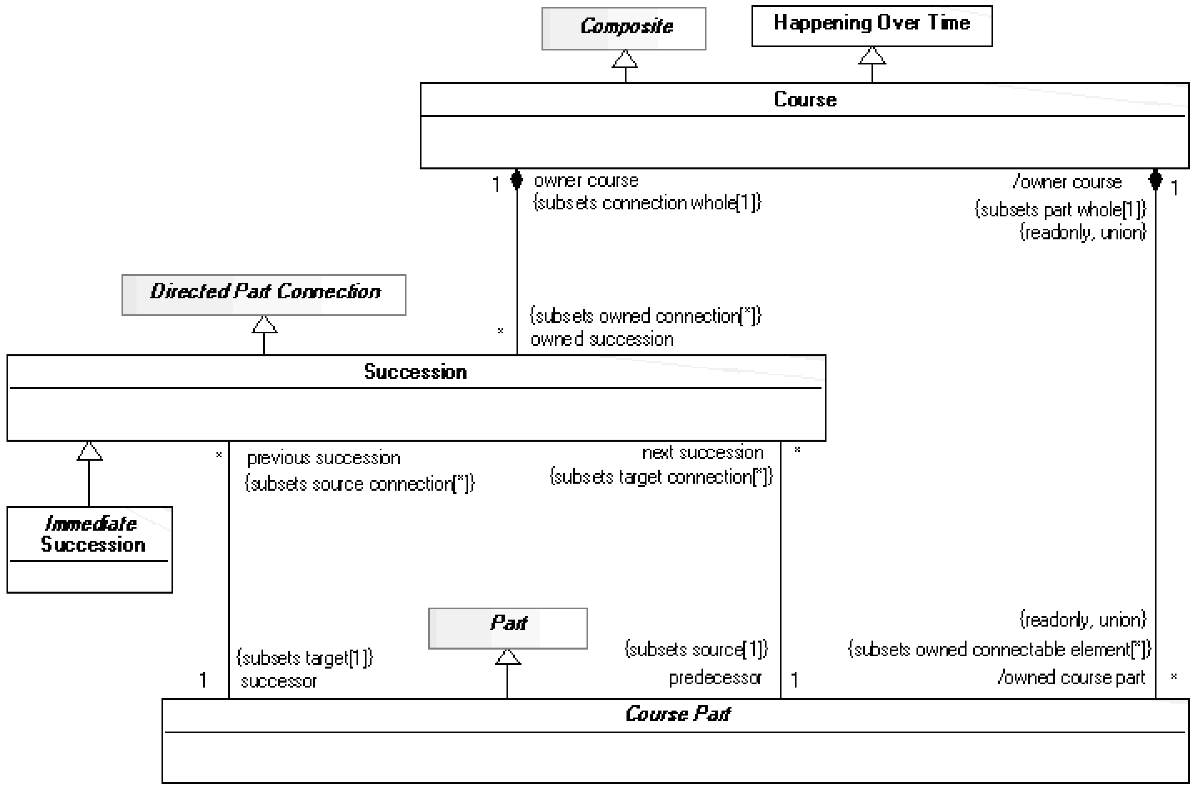


Figure 57 - Course Diagram

4.5.2.7 Gateway Diagram

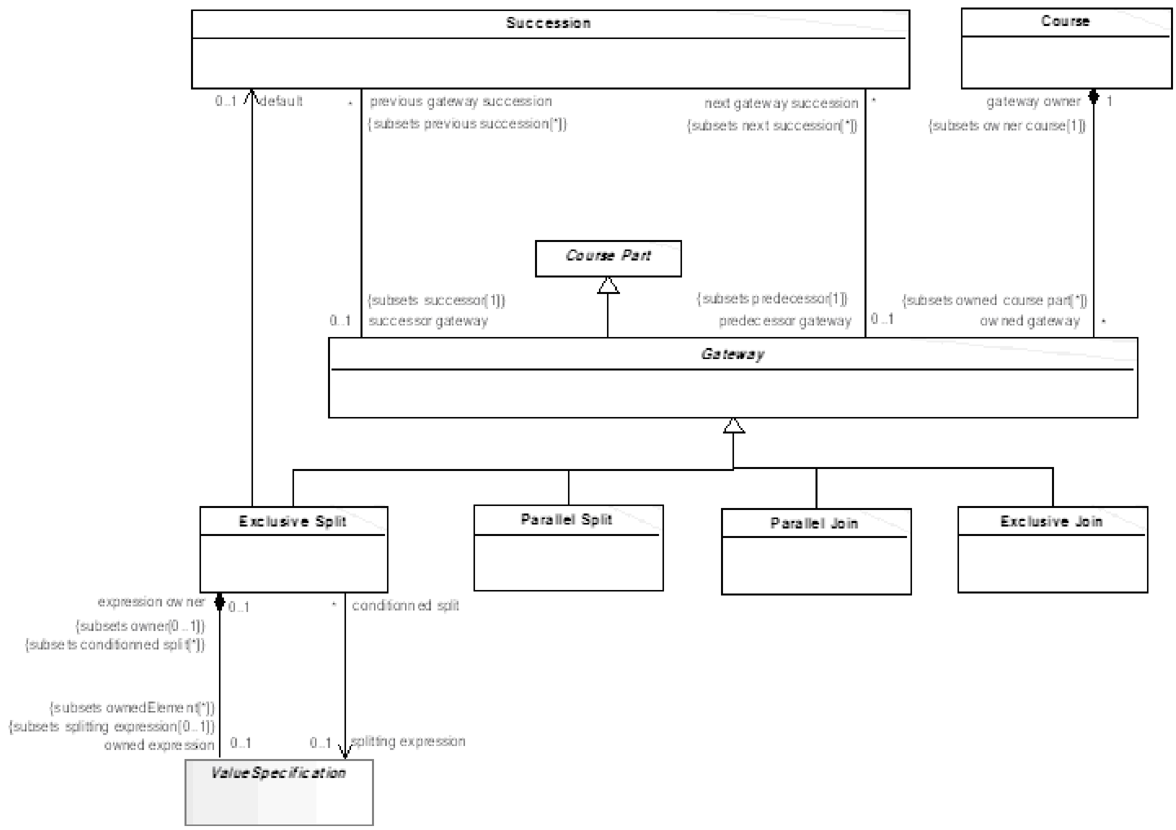


Figure 58 - Gateway Diagram

4.5.2.8 Event Course Diagram

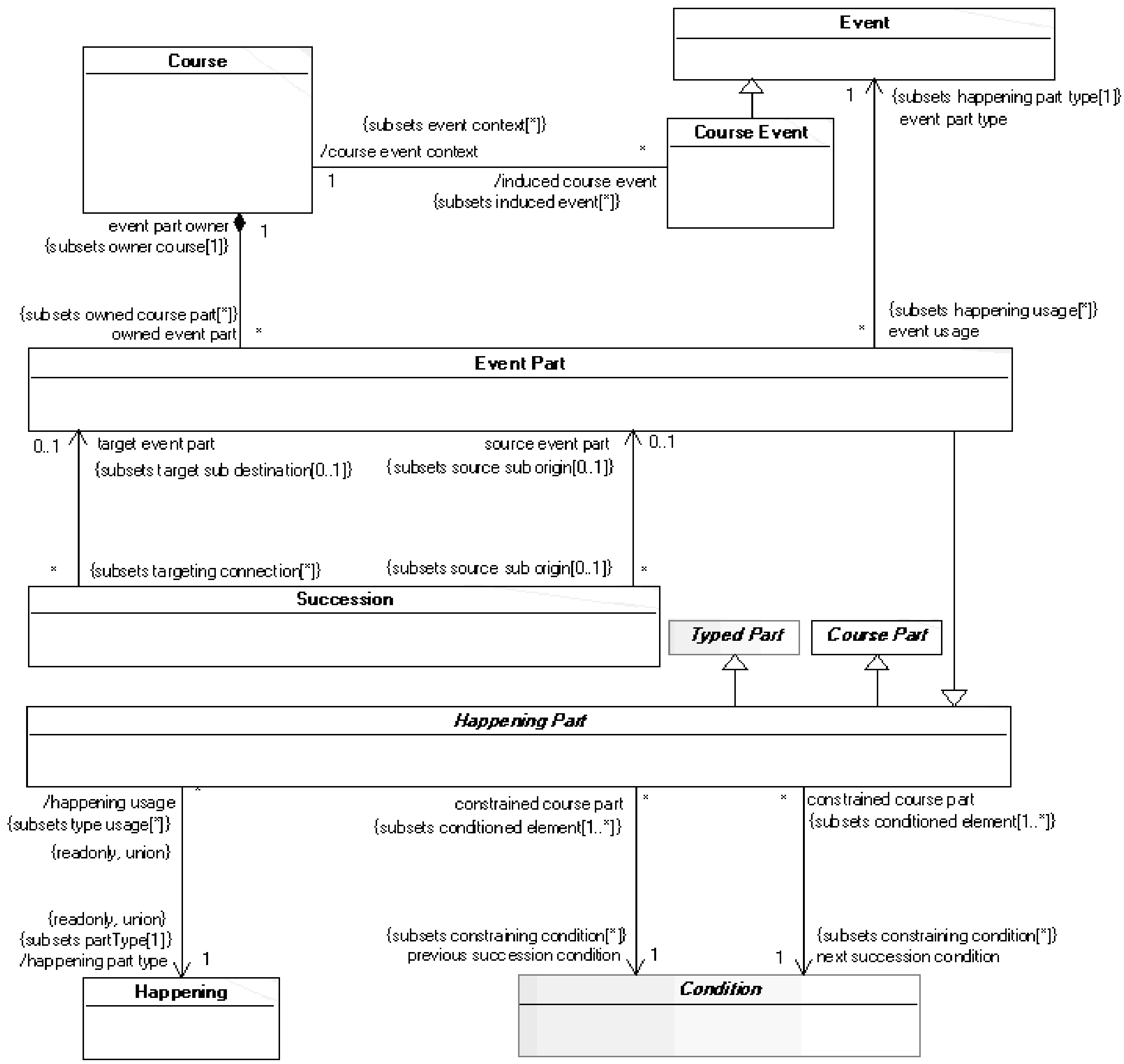


Figure 59 - Event Course Diagram

4.5.2.9 Common Infrastructure Library: Happenings, Events and Conditions

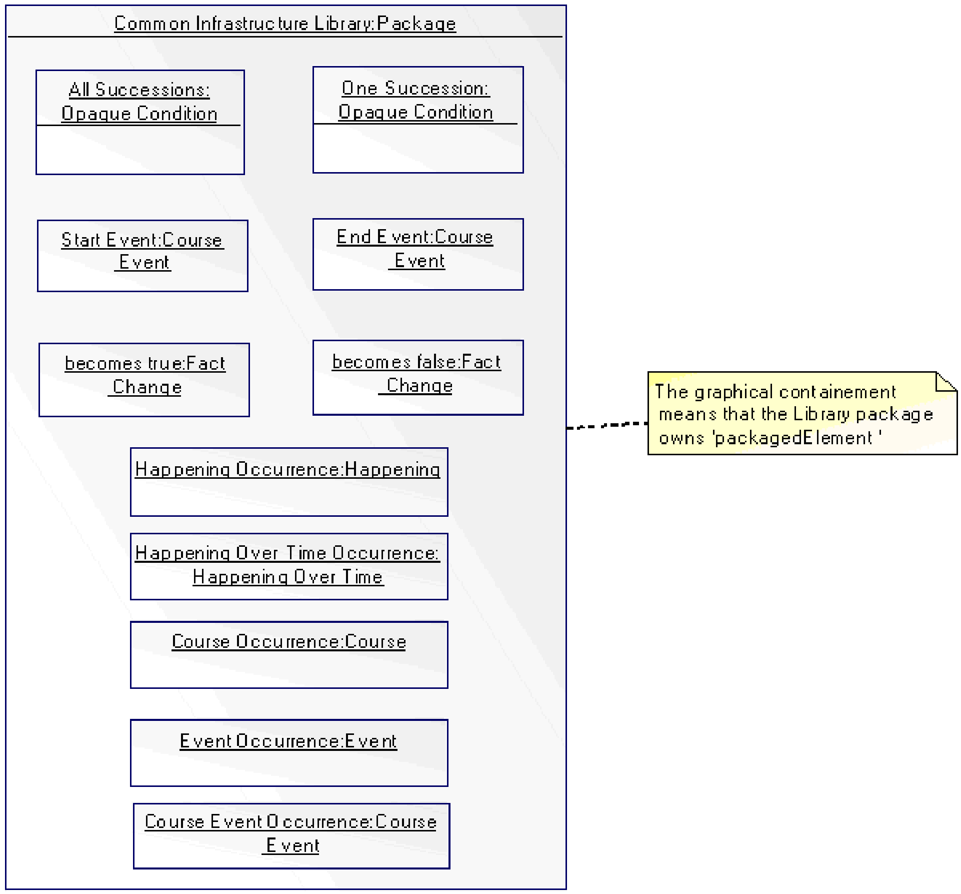


Figure 60 - Common Infrastructure Library: Happenings, Events and Conditions

4.5.2.10 Common Infrastructure Library: 'Happening Occurrences'

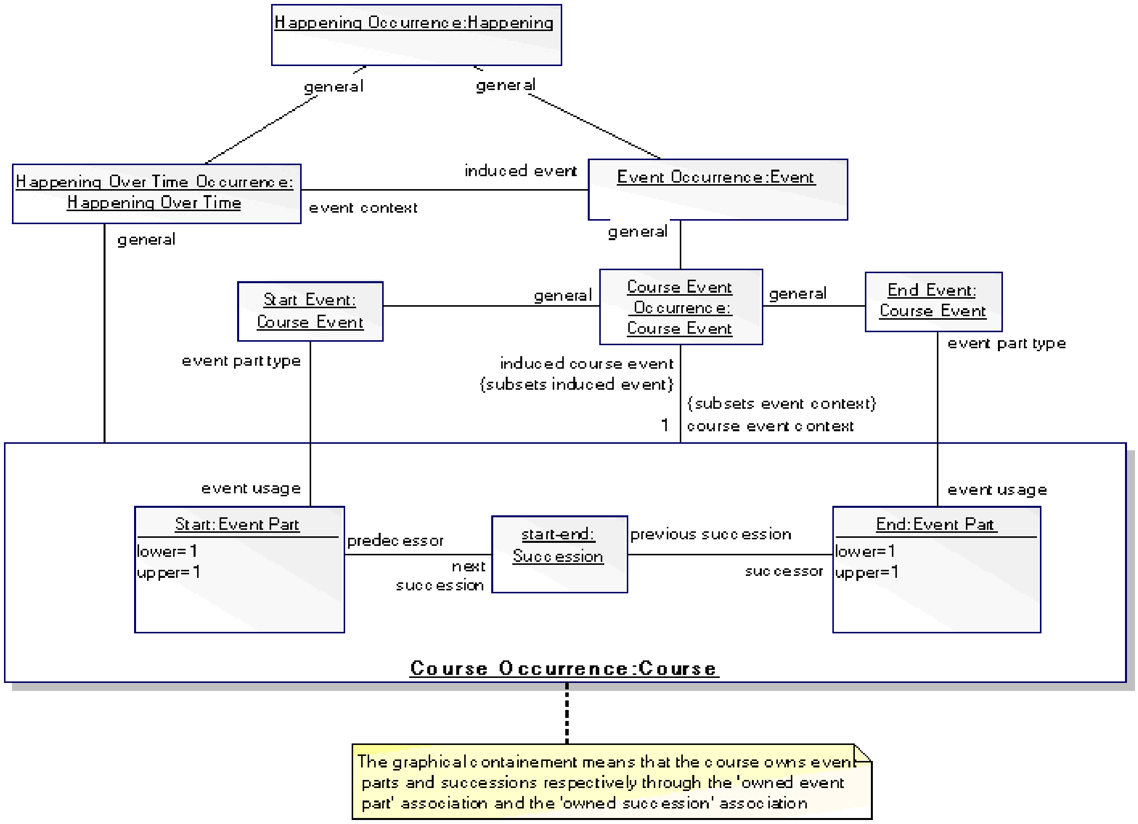


Figure 61 - Common Infrastructure Library: 'Happening Occurrences'

4.5.2.11 Clock

Package: Course Model
isAbstract: No
Generalization: “Happening Over Time”

Description

A **Clock** is a kind of **Happening Over Time** that produces **Time Events**.

Associations

produced time event : Time Event [*]	Time Event that occurs in the context of a Clock Subsets *induced event*

4.5.2.12 Course

Package: Course Model
isAbstract: No
Generalization: “Composite” “Happening Over Time”

Description

A **Course** is an ordered **Succession** of **Happening Part**s
A **Course** is a **Composite** that has connections representing that one part of the course "follows" another in time, and possibly establishes constraints on such followings (**Succession**).

Associations

induced course event : Course Event [*]	Events that can occur in the context of this Course. The set of these Events is derived from the Event Part owned by the Course. This is a derived association. Subsets *induced event*
owned course part : Course Part [*]	Course Part owned by the Course This is a derived union. Subsets *owned connectable element*
owned event part : Event Part [*]	Event Part owned by the Course Subsets *owned course part*
owned gateway : Gateway [*]	Gateway owned by the Course. Subsets *owned course part*
owned succession : Succession [*]	Succession owned by the Course Subsets *owned connection*

4.5.2.13 Course Event

Package: Course Model
isAbstract:
Generalization: “Event”

Description

A **Course Event** is a kind of **Event** that occurs as part of the lifecycle of a **Course**, such as **Start Event**, **End Event**. The **Common Infrastructure** provides a predefined library of **Course Events**.

Associations

course event context : Course [1]	Event that can occur in the context of the Course This is a derived association. Subsets *event context*

4.5.2.14 Course Part

Package: Course Model
isAbstract: Yes
Generalization: “Part”

Description

A **Course Part** is a kind of **Connectable Element** that defines a stage in a **Course**. It can be connected to **Succession** as a **predecessor** or **successor** element.

Associations

next succession : Succession [*]	Succession that enables the Course Part as its predecessor . Subsets *target connection*
previous succession : Succession [*]	Succession that enables the Course Part as its successor . Subsets *source connection*

4.5.2.15 Cycle Event

Package: Course Model
isAbstract: No
Generalization: “Time Event”

Description

A **Cycle Event** is a kind of **Time Event** that define the occurrence of a cycle in time.

Attributes

timedatePeriod: UnlimitedNatural [1]

4.5.2.16 Event

Package: Course Model
isAbstract: No
Generalization: “Happening”

Description

An **Event** is a **Happening** for dynamic entities occurring at a point in time.

Associations

event context : Happening Over Time [*]	Happening Over Time where the Event can occur

4.5.2.17 Event Condition

Package: Course Model
isAbstract: Yes
Generalization: “Condition”

Description

An **Event Condition** is a **Condition** for specifying that an **Event** must occur in the context of a particular **Happening Over Time** for the condition to hold. For instance, a condition can be on the eruption (instance of **Event**) of a particular volcano (instance of **Happening Over Time**).

Associations

conditioning event : Event [1]	Event that is the source of the Event Condition. This is a derived union.
conditioning happening over time : Happening Over Time [0..1]	Happening Over Time where the conditioning event should occur. This is a derived union.

4.5.2.18 Event Part

Package: Course Model
isAbstract:
Generalization: "Happening Part"

Description

An **Event Part** identifies **Event** (such as **Start Event** or **End Event**) for an individual **Course**. An **Event Part** is also a **Happening Part**, enabling it to be connected by **Successions**.

Associations

event part type : Event [1]	Event that is the type of the Event Part. Subsets *happening part type*

4.5.2.19 Exclusive Join

Package: Course Model
isAbstract: No
Generalization: "Gateway"

Description

An **Exclusive Join** is a **Gateway** indicating that the part following it will occur after each part that occurs preceding it.

BPMN Notation

The Exclusive Join shares the same basic shape of the generic **Gateway**.

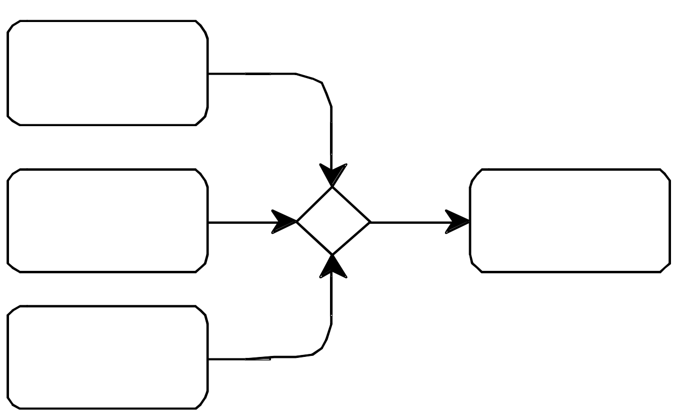


Figure 62 - Exclusive Merge Notation

4.5.2.20 Exclusive Split

Package: Course Model
isAbstract: No
Generalization: “Gateway”

Description

Exclusive Split is a **Gateway** indicating that exactly one of the parts following it will occur after the part preceding it.

Associations

default : Succession [0..1]	Succession enabled by default if no other next succession connected to the Exclusive Split has been enabled.
owned expression : ValueSpecification [0..1]	splitting expression owned by the Exclusive Split. Subsets *ownedElement* Subsets *splitting expression*
splitting expression : ValueSpecification [0..1]	ValueSpecification that specifies the expression shared by the guards on the outgoing successions of the Exclusive Split .These guards must be Fact Conditions that reference this shared ValueSpecification as their evaluated expression.

Constraint

The **guard s** of the **next succession s** of the **Exclusive Split** must be **Fact Condition**s that have their **evaluated expression** be the same as the **splitting expression** of the **Exclusive Split**.

self.**next succession** ->**guard** ->**evaluated expression** in self. **splitting expression**

[1] The **default Succession** must be one of the **Successions** connected to the **Exclusive Split** as a **next succession**.

BPMN Notation

The Exclusive Split shares the same basic shape, called a Gateway, of the generic **Gateway**. The Exclusive Split MAY use a marker that is shaped like an “X” and is placed within the Gateway diamond to distinguish it from other **Gateways**. This marker is not required. A Diagram SHOULD be consistent in the use of the “X” internal indicator. That is, a Diagram SHOULD NOT have some Exclusive Splits with an indicator and some Exclusive Splits without an indicator.

The **default** succession is represented by a default Marker that MUST be a backslash near the beginning of the line representing the **Succession**.

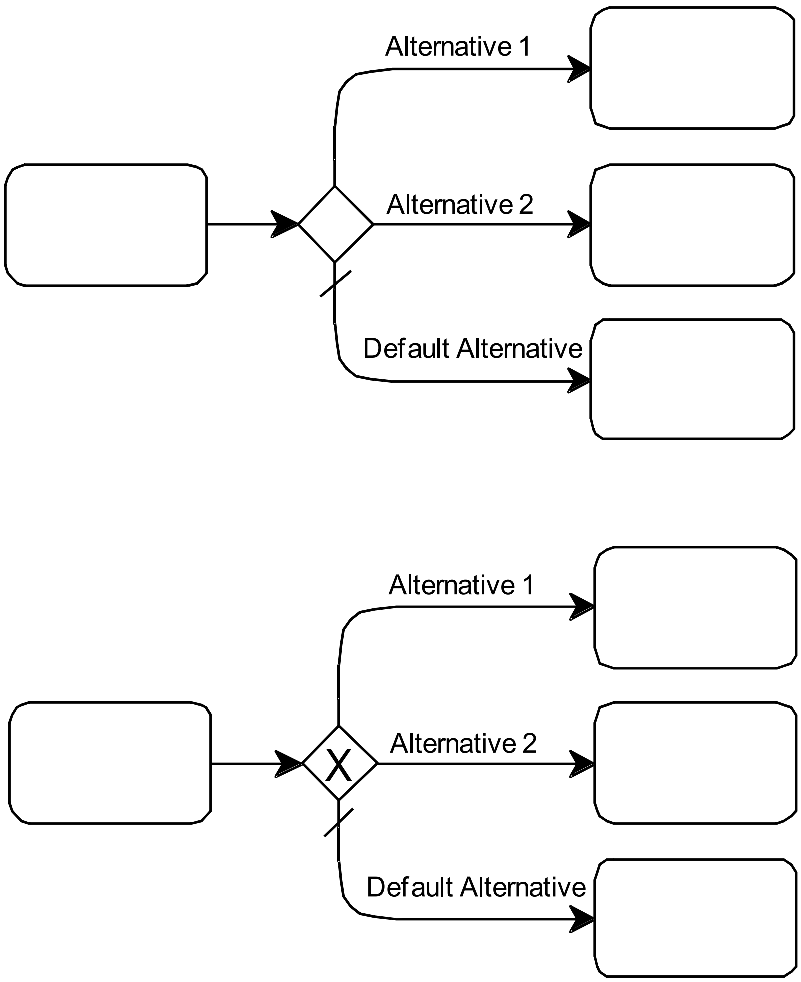


Figure 63 - Exclusive Split Notation

4.5.2.21 Fact Change

Package: Course Model
isAbstract: No
Generalization: “Event”

Description

A **Fact Change** is a kind of **Event** that manifests a change in the evaluation of a **Statement**.

BPMN Notation

Figure 64 - Fact Change Notation

4.5.2.22 Fact Change Condition

Package: Course Model
isAbstract: No
Generalization: "Event Condition"

Description

A **Fact Change Condition** refers to general propositions becoming true or false due to changes in M0 facts. It is used to integrate with models of rules.

Associations

conditioning fact change : Fact Change [1]	Fact Change that, when it occurs, make the Fact Change Condition evaluate to true Subsets *conditioning event*
conditioning statement : Statement [1]	Statement that the Fact Change Condition is evaluating the change of.

4.5.2.23 Gateway

Package: Course Model
isAbstract: Yes
Generalization: "Course Part"

Description

A **Gateway** is a kind of **Course Part** representing potentially complex specifications of how dynamic individuals playing **Happening Part**s are ordered in time. The particular specifications are given in subtypes. At runtime, **Gateways** don't have any execution trace.

Associations

next gateway succession : Succession [*]	Succession that enables the Gateway as its predecessor gateway. Subsets *next succession*
previous gateway succession : Succession [*]	Succession that enables the Gateway as its successor gateway. Subsets *previous succession*

BPMN Notation

A Gateway is represented by a diamond that has been used in many flow chart notations for exclusive branching and is familiar to most modelers. The diamond MUST be drawn with a single thin black line. It is not a requirement that predecessor and successor Successions must connect to the corners of the diamond. Successions can connect to any position on the boundary of the Gateway.

The shape of the different sub-types of Gateway are differentiated by an internal marker. This marker MUST be placed inside the shape, in any size or location, depending on the preference of the modeler or modeling tool vendor.

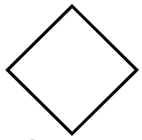


Figure 65 - Gateway Notation

4.5.2.24 Happening

Package: Course Model
isAbstract: No
Generalization: “Classifier”

Description

A **Happening** is a **Classifier** for dynamic entities.

4.5.2.25 Happening Over Time

Package: Course Model
isAbstract: No
Generalization: “Happening”

Description

A **Happening Over Time** is a **Happening** for dynamic entities that are treated as extending over time and that are contexts for **Events**.

Associations

induced event : Event [*]	Event that occurs in the context of the Happening Over Time

4.5.2.26 Happening Part

Package: Course Model
isAbstract: Yes
Generalization: “Course Part” “Typed Part”

Description

A **Happening Part** is a kind of **Course Part** that is also a **Typed Part** where the type is a **Happening**. It is a stage or interval in a development or Course.

Happening Parts are different from other **Course Parts** as they are the only one that have occurrence trace at runtime.

Associations

happening part type : Happening [1]	Happening that is the type of the Happening Part. This is a derived union. Subsets *partType*

next succession condition : Condition [1]	conditions next succession (outgoing) must satisfy when dynamic entities playing a part come to an end. Subsets *constraining condition* Default: All Successions
previous succession condition : Condition [1]	condition previous succession (incoming) must satisfy for dynamic entities playing a part to start, Subsets *constraining condition* Default: One Succession

4.5.2.27 Immediate Succession

Package: Course Model
isAbstract: Yes
Generalization: "Succession"

Description

A **Immediate Succession** is a kind of **Succession** that has the following execution semantic: **successor** immediately follows its **predecessor**.

4.5.2.28 Parallel Join

Package: Course Model
isAbstract: No
Generalization: "Gateway"

Description

Parallel Join is a **Gateway** indicating that the parts (in the sense of individuals) following it happen after the parts preceding them.

BPMN Notation

The Parallel Join uses the shape of **Gateway**, called Gateway and MUST use a marker that is in the shape of a plus sign and is placed within the Gateway diamond to distinguish it from other **Gateways**.

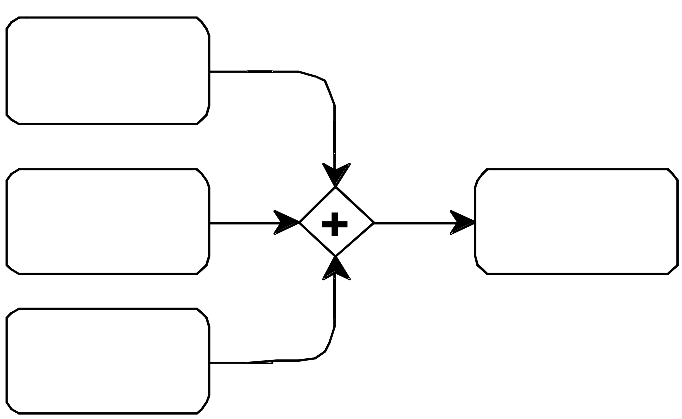


Figure 66 - Parallel Join Notation

4.5.2.29 Parallel Split

Package: Course Model
isAbstract: No
Generalization: “Gateway”

Description

Parallel Split is a **Gateway** that indicates that the dynamic individuals playing parts following them happen after the dynamic individuals playing the part preceding them.

BPMN Notation

The Parallel Split uses the shape of **Gateway**, called Gateway and MUST use a marker that is in the shape of a plus sign and is placed within the Gateway diamond to distinguish it from other **Gateways**.

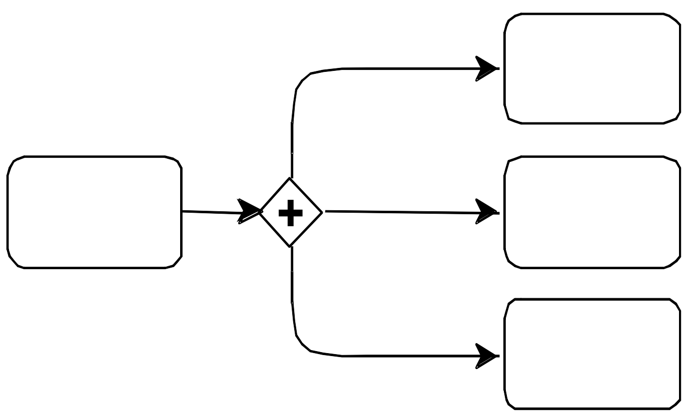


Figure 67 - Parallel Split Notation

4.5.2.30 Relative TimeDate Event

Package: Course Model
isAbstract: No
Generalization: “Time Event”

Description

A **Relative TimeDate Event** is a kind of **TimeDate Event** that defines a change in time for a relative start point in time.

Attributes

duration: UnlimitedNatural [1]

Associations

starting event : Event [1]	Event which occurrence is the beginning of the Relative TimeDate Event

4.5.2.31 Succession

Package: Course Model
isAbstract: No
Generalization: "Directed Part Connection"

Description

A **Succession** is a **Directed Part Connection** that organizes **Course Parts** in series in the context of a **Course**. A **Succession** indicates that one **Course Part** "follows" another in time, and possibly establishes constraints on such followings. It can order the **Event Part** of its **Happening Parts** such as their **Start** or **End**.

Succession allows any combination of **Event Part** to be connected.

End -> Start
Start -> Start
Start -> Abort
etc.

A **Succession** doesn't need to have **Happening Part** on its ends, it can have untyped course parts also, such as **Gateway**, but it must have something on each end.

For convenience, a **Succession** that does not specify **source event part** or **target event part** will have the same effect as a **Succession** where these are respectively the **End** and **Start**.

Associations

predecessor gateway : Gateway [0..1]	Gateway that comes before another Course Part in a Succession. Subsets *predecessor*
predecessor : Course Part [1]	Course Part that comes before another Course Part in a Succession. Subsets *source*
source event part : Event Part [0..1]	Event Part of the predecessor Happening Part that is connected through the Succession. Subsets *source sub origin*
successor gateway : Gateway [0..1]	Gateway that comes after another Course Part in a Succession. Subsets *successor*
successor : Course Part [1]	Course Part that comes after another Course Part in a Succession. Subsets *target*
target event part : Event Part [0..1]	Event Part of the successor Happening Part that is connected through the Succession. Subsets *target sub destination*

Constraint

[1] The **source event part** must be one of the **Event**s of the **Course** that is the type of the **predecessor**.

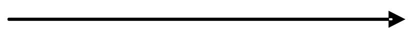
processing step self.**source event part** in self.**predecessor behavioral step**->**step type** ->**owned event part**

[1] The **target event part** must be one of the **Event**s of the **Course** that is the type of the **successor processing**.

step self.**target event part** in self.**successor behavioral step->step type ->owned event part**

BPMN Notation

A Succession is line with a solid arrowhead that MUST be drawn with a solid single line

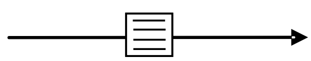


A succession

Figure 68 - Succession Notation

Non Normative Notation

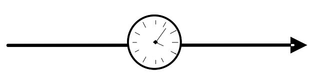
A Succession with a **Condition** of type **Fact Change Condition is** drawn as a line covered by the shape the conditioning **Fact Change**. The line has a solid arrowhead and MUST be drawn with as solid single line.



A succession with Fact Change Condition

Figure 69 - Succession with Fact Change Condition

A Succession with a **Condition** of type **Time Event Condition** is drawn as one line covered by the shape the conditioning **Time Event**. The line has a solid arrowhead and MUST be drawn with a solid single line.



A succession with Time Change Condition

Figure 70 - Succession with Time Event Condition

4.5.2.32 Time Event

Package: Course Model
isAbstract: No
Generalization: "Event"

Description

A **Time Event** specifies a point in time that is a source of interest.

Attributes

timeExpression: String [0..1]	A timeExpression represents a time value.

Associations

time event producer : Clock [0..1]	Clock that generates the Time Event Subsets *event context*

BPMN Notation

A **Time Event** is represented by a clock

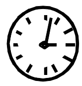


Figure 71 - Time Event Notation

4.5.2.33 Time Event Condition

Package: Course Model
isAbstract: No
Generalization: "Event Condition"

Description

A **Time Event Condition** is a kind of **Event Condition** that is based on the occurrence of a **Time Event**. A **Time Event Condition** is specified by referring to a **Clock**.

Associations

conditioning clock : Clock [0..1]	Clock that is the Happening Over Time context producing the conditioning time event that is the source of the Time Event Condition. Subsets *conditioning happening over time*
conditioning time event : Time Event [1]	Time Event that is the source of the Time Event Condition. Subsets *conditioning event*

4.5.2.34 TimeDate Event

Package: Course Model
isAbstract: No
Generalization: "Time Event"

Description

A **TimeDate Event** is a kind of **Time Event** that manifest a date or time change.

Attributes

timedate: UnlimitedNatural [1]

4.5.2.35 Instance: All Successions

Class: Opaque Condition

Description

Condition requiring all successions to be satisfied before the execution of a **Happening Part**.

Links

Played End	*Opposite End*
All Successions:owningPackage	*owningPackage* Common Infrastructure Library

4.5.2.36 Instance: becomes false

Class: Fact Change

Description

Links

Played End	*Opposite End*
becomes false:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.37 Instance: becomes true

Class: Fact Change

Description

Links

Played End	*Opposite End*
becomes true:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.38 Instance: Course Event Occurrence

Class: Course Event

Description

Links

Played End	*Opposite End*
Course Event Occurrence:	*general* Event Occurrence
Course Event Occurrence:general	Start Event
Course Event Occurrence:general	End Event
Course Event Occurrence:packagedElement	*owningPackage* Common Infrastructure Library
Course Event Occurrence:induced course event	*course event context* Course Occurrence

4.5.2.39 Instance: Course Occurrence

Class: Course

Description

Course Occurrence is a **Course** that is the generalization of all M1 **Courses**, including all orchestrations and choreographies.

Course Occurrence introduces M1 events for starting and ending and a succession between them that is inherited to all M1 courses. All individual (M0) courses conform to **Course Occurrence**, which is the most abstract M1 model of **Courses**.

Links

Played End	*Opposite End*
Course Occurrence:	*general* Happening Over Time Occurrence
Course Occurrence:course event context	*induced course event* Course Event Occurrence
Course Occurrence:event part owner	*owned event part* End
Course Occurrence:event part owner	*owned event part* Start
Course Occurrence:general	Behavior Occurrence
Course Occurrence:owner course	*owned succession* start-end
Course Occurrence:packagedElement	*owningPackage* Common Infrastructure Library

Constraint

[1] Start and End event parts cannot have the same values
not self.Start = self.End

Non Normative Notation

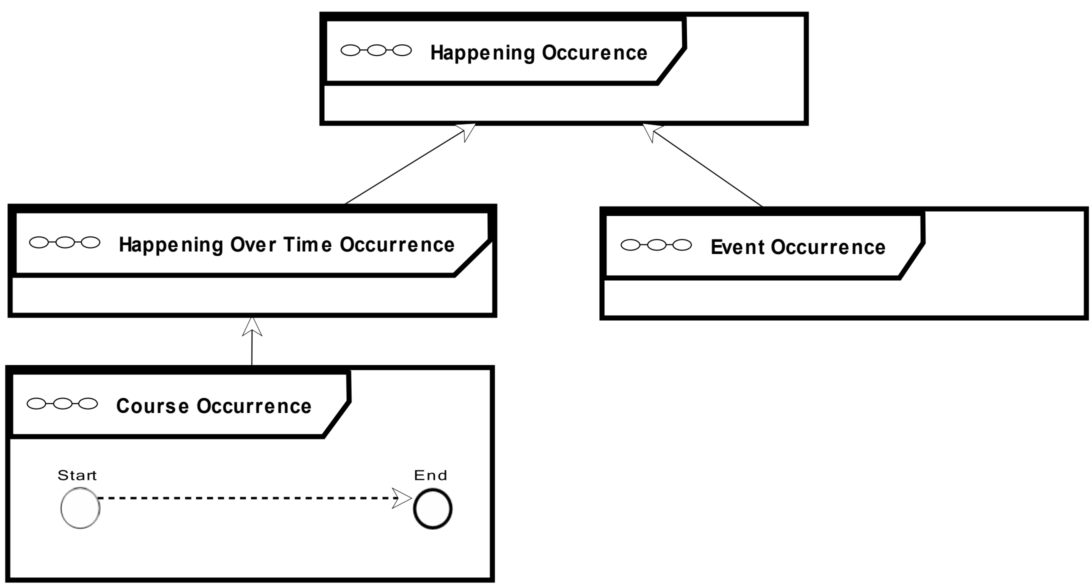


Figure 72 - Course Occurrence Diagram

4.5.2.40 Instance: End Event

Class: Course Event

Description

End Event is a **Event** that manifests the end of a **Course**.

Links

Played End	*Opposite End*
End Event:	*general* Course Event Occurrence
End Event:event part type	*event usage* End
End Event:general	Normal End Event
End Event:general	Abnormal End Event
End Event:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.41 Instance: End

Class: Event Part

Description

Links

Played End	*Opposite End*
End:event usage	*event part type* End Event
End:owned event part	*event part owner* Course Occurrence
End:subsettedProperty	Abnormal End
End:subsettedProperty	Normal End
End:successor	*previous succession* interationend-end
End:successor	*previous succession* startseq-end
End:successor	*previous succession* compensate-end
End:successor	*previous succession* cancel-end
End:successor	*previous succession* start-end

BPMN Notation

The shape of the **End** instance of **Event Part** is drawn as a circle that MUST be drawn with a single thick black line.

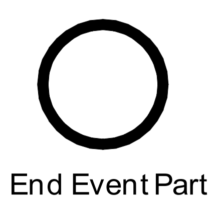


Figure 73 - Event Part : End Notation

4.5.2.42 Instance: Event Occurrence

Class: Event

Description

Event Occurrence is an Event that is the generalization of all M1 events, including all events induced by orchestrations and choreographies. All individual (M0) occurrences of events conform to Event Occurrence, which is the most abstract M1 model of events.

Links

Played End	*Opposite End*
Event Occurrence:	*general* Happening Occurrence
Event Occurrence:general	Course Event Occurrence
Event Occurrence:induced event	*event context* Happening Over Time Occurrence
Event Occurrence:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.43 Instance: Happening Occurrence

Class: Happening

Description

Happening Occurrence is a **Happening** that is the generalization of all M1 happenings over time and events, including all orchestrations and choreographies and events induced by them. All individual (M0) occurrences of happenings over time and events conform to Happening Occurrence, which is the most abstract M1 model of occurrence.

Links

Played End	*Opposite End*
Happening Occurrence:general	Event Occurrence
Happening Occurrence:general	Happening Over Time Occurrence

4.5.2.44 Instance: Happening Over Time Occurrence

Class: Happening Over Time

Description

Happening Over Time Occurrence is a Happening Over Time that is the generalization of all M1 happenings over time, including all orchestrations and choreographies. All individual (M0) happening of time occurrences conform to Happening Over Time Occurrence, which is the most abstract M1 model of happening over time.

Links

Played End	*Opposite End*
Happening Over Time Occurrence:	*general* Happening Occurrence
Happening Over Time Occurrence:event context	*induced event* Event Occurrence
Happening Over Time Occurrence:general	Course Occurrence
Happening Over Time Occurrence:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.45 Instance: One Succession

Class: Opaque Condition

Description

Condition requiring only one succession to be satisfied before the execution of a **Happening Part**.

Links

Played End	*Opposite End*
One Succession:owningPackage	*owningPackage* Common Infrastructure Library

4.5.2.46 Instance: Start Event

Class: Course Event

Description

Start Event is a **Event** that manifests the start of a **Course**

Links

Played End	*Opposite End*
Start Event:	*general* Course Event Occurrence
Start Event:event part type	*event usage* Start
Start Event:packagedElement	*owningPackage* Common Infrastructure Library

4.5.2.47 Instance: start-end

Class: Succession

Description

Links

Played End	*Opposite End*
start-end:next succession	*predecessor* Start
start-end:owned succession	*owner course* Course Occurrence
start-end:previous succession	*successor* End

4.5.2.48 Instance: Start

Class: Event Part

Description

Links

Played End	*Opposite End*
Start:event usage	*event part type* Start Event
Start:owned event part	*event part owner* Course Occurrence
Start:predecessor	*next succession* start-iterationend
Start:predecessor	*next succession* start-cancel
Start:predecessor	*next succession* start-end
Start:predecessor	*next succession* start-compensate
Start:target event part	start/start

BPMN Notation

An **Event Part** typed by the **Start Event** instance of **Event** is drawn as a circle that MUST be drawn with a single thin line.

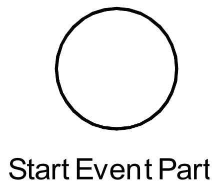


Figure 74 - Event Part : Start Notation

When a **Start Event Event Part** is conditioned by a **Fact Change Condition**, a **Fact Change** marker is added to the **Start Event Event Part** shape.

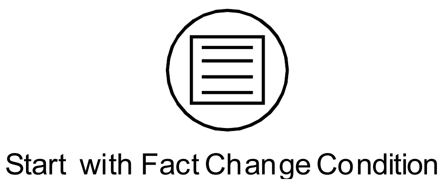


Figure 75 - Event Part : Start with 'Fact Change Condition' Notation

Shape of **Start** when it has an **Event Monitor** with a **Time Event Condition**, as its predecessor.

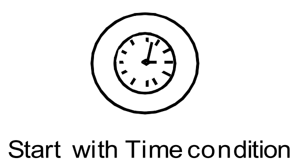


Figure 76 - Event Part : Start with 'Time Event Condition' Notation

Index