ICS 35.080
L 77

中华人民共和国国家标准

GB/T 28174.1—2011

统一建模语言(UML)
第1部分:基础结构

Unified modeling language(UML)—
Part 1:Infrastructure

2011-12-30 发布　　2012-06-01 实施

中华人民共和国国家质量监督检验检疫总局
中国国家标准化管理委员会　发布

目　　次

前　言

GB/T 28174《统一建模语言(UML)》分为4个部分：

——第1部分：基础结构；

——第2部分：上层结构；

——第3部分：对象约束语言(OCL)；

——第4部分：图交换。

本部分为GB/T 28174的第1部分。

本部分按照GB/T 1.1—2009给出的规则起草。

本部分参考面向对象工作组(OMG)的《统一建模语言：基础结构》2.0版。

请注意本文件的某些内容可能涉及专利。本文件的发布机构不承担识别这些专利的责任。

本部分由全国信息技术标准化技术委员会(SAC/TC 28)提出并归口。

本部分起草单位：北京大学、广东省广业信息产业集团有限公司、广东万维博通信息技术有限公司、中国电子技术标准化研究所。

本部分主要起草人：麻志毅、许立勇、周伟强、唐泽欢、江善东、高健。

引　言

统一建模语言(UML)是一种可视化规约语言,用于定义和构造计算机信息系统的制品,并将其文档化。它是一种通用建模语言,可以和所有主流的面向对象和面向构件的方法一起使用,并适用于所有的应用领域和实现平台(如:CORBA、J2EE、.NET 等)。

0.1　统一建模语言不同版本之间的关系

由于 UML 的技术较新,所以该国际标准历经多次的版本演化,下面是 UML 在 OMG 的演化过程:

1997　　UML1.1
1998　　UML1.2
1999　　UML1.3
2001　　UML1.4
2003　　UML2.0

GB/T 28174 的本部分正文中的 UML 均指 UML2.0 统一建模语言和 GB/T 28174。

0.2　关于对读者的建议

需要了解语言中的元模型构造物,利用这些构造物进行元模型扩展或者是构造新的建模语言的用户可阅读基础结构部分(GB/T 28174.1)。

应用系统建模用户和建模工具制造方都需阅读上层结构(GB/T 28174.2)。但要注意,该部分的内容是交叉引用的,可不按目次顺序阅读。

对于要精确地对模型进行约束的应用系统建模用户或要支持对象约束语言的建模工具制造方,需阅读对象约束语言部分(GB/T 28174.3)。

支持在不同的软件工具间平滑且无缝地交换文档的建模工具制造方,需阅读图交互部分。

0.3　关于本部分

本部分的第 4 章和第 5 章描述了定义 UML 语言体系结构和规格说明的方法。

本部分的第 6 章至第 10 章描述了元模型的基础结构库(Infrastructure Library)的结构和内容,这些元模型包括 UML 元模型和相关元模型,如元对象设施(MOF)和公共仓库元模型(CWM)。基础结构库定义了 UML 的可重用元语言核心与元模型扩展机制。元语言核心能够用于制定各种元模型,包括 UML、MOF 和 CWM。另外,基础结构库还定义了一种外廓扩展机制,当某些平台和建模领域不具备元模型建模能力时,利用这种扩展机制可以为这些平台对 UML 进行定制。基础结构库的最顶层包如图 1 所示。

核心包是基础结构库可重用部分的主体,而且被进一步细分,如图 3 所示。

原子类型(PrimitiveTypes)包比较简单,它包含若干预定义类型,预定义类型通常用于元模型(metamodeling)建模,因此它们不但用于基础结构库本身,而且用于 MOF 和 UML 等元模型(metamodels)。抽象包包括若干只含有少量元模型且粒度适当的包,它们中的大部分是抽象的。这个包的目的是提供高可用的元类集,在定义元模型时被特化。构造包也包含若干粒度适当的包,且把抽象包的多个方面集中在一起。构造包中的元类趋向于具体而不是抽象,并且适用于面向对象建模范式。来看一下元模型,如 MOF 和 UML,它们通常因为要自动输入核心中其他包的内容而引入构造包。基本包(Basic)包括一个构造包的子集,它主要是为了使用 XMI。

外廓包(profiles)包含创建特定元模型外廓的机制,尤其是对 UML 的扩展。这种扩展机制是 MOF 提供的通用扩展功能的子集。

统一建模语言(UML)
第1部分:基础结构

1 范围

GB/T 28174的本部分规定了用于对各类软件系统进行可视化、详述、构造和文档化的统一建模语言。本语言也可用于对其他领域进行建模。

本部分适用于统一建模语言(UML)的基础语言构造物,包括讲述UML的体系结构、UML的设计原理以及如何应用这些原理来组织UML的方法。

2 规范性引用文件

下列文件对于本文件的应用是必不可少的。凡是注日期的引用文件,仅注日期的版本适用于本文件。凡是不注日期的引用文件,其最新版本(包括所有的修改单)适用于本文件。

GB/T 28174.2 统一建模语言(UML) 第2部分:上层结构

GB/T 28174.3 统一建模语言(UML) 第3部分:对象约束语言(OCL)

GB/T 28174.4 统一建模语言(UML) 第4部分:图交换

3 术语和定义、缩略语

3.1 术语和定义

下列术语和定义适用于本文件,也适用于GB/T 28174.2、GB/T 28174.3和GB/T 28174.4。

3.1.1

抽象类 abstract class

不能直接被实例化的类。

相对语:**具体类**(3.1.49)

3.1.2

抽象 abstractio

强调事物的一定特征而忽视无关的其他特征的结果。所定义的抽象与观察者的视角有关。

3.1.3

动作 action

行为规约的基础单元,用以描述所建模系统(计算机系统或现实世界系统)中的转换或处理。动作包含在活动中,活动提供动作的语境。

见:**活动**(3.1.9)。

3.1.4

动作序列 action sequence

解析为一系列动作的表达式。

3.1.5

动作状态 action state

表示原子动作执行的状态,通常为操作调用。

3.1.6

激活　activation

动作执行的启动。

3.1.7

主动类　active class

其实例为主动对象的类。

见：**主动对象**(3.1.8)。

3.1.8

主动对象　active object

可以执行其自己的行为而不要求方法调用的对象。有时把它称为“拥有控制线程的对象”。主动对象响应来自其他对象的通信点，由主动对象的行为单独决定，而不是由调用对象决定。这暗示着主动对象在一定程度上是自治的和交互式的。

见：**主动类**(3.1.7)和(**控制**)**线程**(3.1.206)。

3.1.9

活动　activity

通过顺序化的从属单元(其基本元素是单个的动作)，表示为执行流的参数化行为的规约。

见：**动作**(3.1.3)。

3.1.10

活动图　activity diagram

使用控制和数据流模型描绘行为的图。

3.1.11

活动者　actor

参与者

在用况中使用的构造物，它定义了当一个用户或任何其他系统与所考虑中的本系统交互时所扮演的一个角色。它是相互作用的实体的类型，但它本身是外部的事物。活动者可表示为人员用户、外部硬件或其他事物。一个活动者不必表示为一个特定的物理实体。例如，单个物理实体可以扮演几个不同的活动者，反过来，单个给定的活动者可以由多个物理实体扮演。

3.1.12

聚集　aggregate

在聚合关系(整体—部分)中作为“整体”的类。

见：**聚合**(3.1.13)。

3.1.13

聚合　aggregation

关联的一种特殊形式，它描述聚集(整体)和部件(部分)间的整体—部分关系。

见：**组合**(3.1.48)。

3.1.14

分析　analysis

系统开发过程的一个阶段，其主要目的是形成独立于实现考虑的问题域模型。分析注重于做什么，设计注重于如何做。

见：**设计**(3.1.63)。

3.1.15

分析时[期]　analysis time

涉及在软件开发过程的分析阶段期间发生的事情。

见：**设计时**(3.1.64)，**建模时**(3.1.120)。

3.1.16

实参　argument

实际参数　actual parameter

对随后要解决的参数的绑定。一个独立的变量。

3.1.17

制品　artifact

产品　product

开发过程所使用或产生的一个物理信息片段。例如，模型、源文件、脚本、二进制可执行文件。可以用制品构成可部署的构件的实现。

相对语：**构件**(3.1.43)。

3.1.18

关联　association

在类目实例间可出现的关系。

3.1.19

关联类　association class

兼有关联和类的性质的模型元素。可以把关联类看作是具有类的性质的关联，或看作具有关联的性质的类。

3.1.20

关联端　association end

关联的端点，它把关联连接到类目。

3.1.21

属性　attribute

类目的结构性特征，它刻画类目的实例。通过命名关系，一个属性把类目的一个实例与一个值或多值联系起来。

3.1.22

辅助类　auxiliary class

一种衍型化的类，通常通过实现从属的逻辑或控制流，它支持另外的更核心的或更基础的类。通常将辅助类与焦点类一起使用，对于在设计阶段规约构件的辅助逻辑和控制流也有一定的作用。

3.1.23

行为　behavior

操作或事件的可观察的效果(包括结果)。它规约产生行为特征的效果的计算。可以采取若干形式来描述行为：交互、状态机、活动或过程(一组动作)。

3.1.24

行为图　behavior diagram

描绘行为特征的一种图形式。

3.1.25

行为特征　behavioral feature

模型元素的动态特征，例如操作或方法。

3.1.26

行为模型侧面　behavioral model aspect

强调系统中的实例行为的模型侧面，包括实例的方法、协作和状态历史。

3.1.27

二元关联　binary association

两个类之间的关联。是 n 元关联的特例。

3.1.28

绑定　binding

通过为模板参数提供实参，从模板创建模型元素。

3.1.29

布尔型　boolean

取值范围为真或假的枚举类型。

3.1.30

布尔表达式　boolean expression

求值为布尔值的表达式。

3.1.31

势　cardinality

集合中的元素个数。

相对语：**势域**(3.1.123)。

3.1.32

子　child

在泛化关系中，另一个元素(父)的特化。

见：**子类**(3.1.193)**和子类型**(3.1.198)。

相对语：**父**(3.1.138)。

3.1.33

调用　call

调用类目上的一个操作的一种动作状态。

3.1.34

类　class

描述一组对象的类目，这些对象共享关于特征、约束和语义的同一规约。

3.1.35

类目　classifier

一组某些方面相同的实例的集合。类目可以具有刻画其实例的特征。类目包含接口、类、数据类型和构件。

3.1.36

分类　classification

实例到类目的指派。

见：**动态分类**(3.1.70)、**多重分类**(3.1.121)、**静态分类**(3.1.185)。

3.1.37

类图　class diagram

显示一组说明性的(静态)模型元素的图，例如，这样的元素可为类、类型以及它们的内容及关系。

3.1.38

客户　client

请求其他类目服务的类目。

相对语：**供方**(3.1.201)。

3.1.39

协作　collaboration

如何实现操作或像用况这样的类目的规约，这样的实现是由用特定的方法扮演特定的角色的一组类目和关联实施的。

见：**交互**(3.1.100)。

3.1.40

协作发生　collaboration occurrence

协作的特殊使用，用以解释一个类目的各部件间或一个操作的各性质间的关系。它也可用以指示协作如何表示类目或操作。一个协作发生指明了一组角色或连接件，按照给定的协作(由协作发生的类型指定)，它们在特定的类目或操作中进行合作。在一个类目或操作中，一个给定的协作可以有多个发生，每一个都涉及一组不同的角色和连接件。一个给定的角色或操作可以出现在同一个或不同协作的多个发生中。

见：**协作**(3.1.39)。

3.1.41

通信图　communication diagram

注重于在生命线间交互的图，在图中描述的核心是内部结构的体系结构以及如何响应传递过来的消息。通过用顺序号的模式给出消息的顺序。顺序图和通信图表达类似的信息，但表示的方式不同。

见：**顺序图**(3.1.175)。

3.1.42

编译时(期)　compile time

涉及到在编译软件模块期间发生的事情。

见：**建模时**(3.1.120)、**运行时**(3.1.170)。

3.1.43

构件　component

系统的模块化部分，它封装自己的内容，且它的声明在其环境中是可以替换的。构件利用提供和请求接口定义自身的行为。这样，构件起类型的作用，其一致性由提供和请求接口来定义(包含静态和动态语义)。

3.1.44

构件图　component diagram

显示构件间的组织和依赖的图。

3.1.45

组合类　composite

一个通过组合关系与一个或多个类发生关系的类。

见：**组合**(3.1.48)。

3.1.46

组合状态　composite state

由并发(正交)子状态或顺序(不相交)子状态组成的状态。

见：**子状态**(3.1.195)。

3.1.47

组合结构图　composite structure diagram

描述类目内部结构，包括该类目与系统其他部分的交互点的图。它图示了共同地执行容器类目的行为的部件的配置。这种体系结构图规约了在特定语境中一组扮演部件(角色)的实例，以及它们所需要的关系。

3.1.48

组合　composition

组成聚合　composite aggregation

聚合的一种形式，它要求部分实例一次最多包含在一个组成类中，组成对象负责创建和销毁其部分。组合可以是递归的。

3.1.49

具体类　concrete class

能直接被实例化的类。

相对语：**抽象类**(3.1.1)。

3.1.50

并发　concurrency

在同一时间段内两个或多个活动的发生。通过交错或同时执行两个或多个线程，实现并发。

见：**线程**(3.1.206)。

3.1.51

并发子状态　concurrent substate

与包含在同一组合状态中的其他子状态同时存在的子状态。

见：**组合状态**(3.1.46)。

相对语：**不相交子状态**(3.1.67)。

3.1.52

可连接元素　connectable element

抽象元类，用以表示可以通过连接件链接的模型元素。

见：**连接件**(3.1.53)。

3.1.53

连接件　connector

使得能够在两个或多个实例间进行通信的链接。可用像指针这样简单的事物或像网络连接这样的复杂事物实现链接。

3.1.54

约束　constraint

语义条件或限制。为了阐述一些模型元素的语义，约束可以用自然语言文本、数学形式化表示法或机器可读的语言来表达。

3.1.55

容器　container

a)　包含其他实例的实例，它提供访问或遍历其内容的操作。如数组、表或集合。

b)　包含其他构件的构件。

3.1.56

包容层次　containment hierarchy

由模型元素以及其间的包容关系组成的命名空间层次。一个包容层次形成一张图。

3.1.57

语境　context

用于特定目的(如规约操作)的一组相关建模元素的视图。

3.1.58

数据类型　data type

其值没有标识的类型，即这样的值是纯值。数据类型包括内建的基本类型(如整型和串)和枚举类型。

3.1.59

委托　delegation

一个对象把消息发给另一个对象让其响应的能力。委托是继承的一种替代方案。

相对语：**继承**(3.1.97)。

3.1.60

依赖　dependency

两个建模元素之间的关系，其中一个建模元素(独立元素)的改变会影响另一建模元素(依赖元素)。

3.1.61

部署图　deployment diagram

描述系统执行的体系结构的图。它把系统制品表示为结点(通过通信路径连接结点能创建具有任意复杂性的网络)。结点通常以嵌套的方式定义，并表示硬件设备或软件执行环境。

见：**构件图**(3.1.44)。

3.1.62

派生元素　derived element

能从其他元素计算出的模型元素，说明它是为了清晰可见，或者说为了设计的目的包含了它，即使它没有添加什么语义信息。

3.1.63

设计　design

系统开发过程的一个阶段，其主要目的是决定怎样实现系统。在设计期间所做的策略和技术决策，用于满足所要求的系统功能需求和质量需求。

3.1.64

设计时(期)　design time

涉及在系统开发过程的设计阶段发生的事情。

见：**建模时**(3.1.120)。

相对语：**分析时**(3.1.15)。

3.1.65

开发过程　development process

在系统开发期间，为特定目的而进行的一组部分有序的步骤，如构造模型或实现模型。

3.1.66

图　diagram

一组模型元素的图形表示，大多数情况下绘制为由弧(关系)和顶点(其他模型元素)组成的连通图。GB/T 28174.2 的附录 A 列出了 UML 所支持的各种图。

3.1.67

不相交子状态　disjoint substate

不能与包容在同一组合状态内的其他子状态同时存在的子状态。

见：**组合状态**(3.1.46)。

3.1.68

分布单元　distribution unit

一组被分配到一个进程或一个处理器的对象或构件集合。可以用运行时组成类或聚集表示分布单元。

3.1.69

域　domain

用一组概念和术语刻画的知识领域或活动领域，由该领域的实践者理解。

3.1.70

动态分类　dynamic classification

一个实例从一个类目到另一个类目的指派。

相对语：**多重分类**(3.1.121)，**静态分类**(3.1.185)。

3.1.71

元素　element

模型的成分。

3.1.72

进入动作　entry action

在一个状态机中，当一个对象进入一个状态时有一个方法执行的动作，不考虑达到该状态所采取的转换。

3.1.73

枚举　enumeration

是一种数据类型，其实例是命名值的列表。例如，RGBColor＝{red，green，blue}。布尔是一种预定义的枚举，其值取自集合{false，true}。

3.1.74

事件　event

对有意义的发生的规约，该发生在时间和空间上有特定位置，并引起相关行为的执行。在状态图的语境中，事件是能触发转换的发生。

3.1.75

异常　exception

一种特殊的信号，通常用以表示故障情形。异常的发送方使执行终止，并且执行由异常的接收者继续，异常的接收者也可能是发送方本身。异常的接收者隐式地由执行期间的交互顺序决定，不显式地指定它。

3.1.76

执行发生　execution occurence

在交互图上表示的生命线中的一种行为单元。

3.1.77

退出动作　exit action

当对象退出状态机的某一状态时由方法执行的动作，而不管退出时所采取的转移。

3.1.78

引出　export

在包的语境中，使某一元素在所处命名空间之外可见。

见：**可见性**(3.1.228)。

相对语：**引入**(3.1.95)。

3.1.79

表达式　expression

计算某一特定类型的值的字符串。例如，表达式“(7＋5＊3)”计算“数”类型的值。

3.1.80

扩展　extend

从扩展用况到基用况的一种关系，它详述了为扩展用况定义的行为如何拓广(遵守在扩展中定义的条件)为基用况定义的行为。该扩展的行为被插入到基用况的扩展点处。基用况不依赖扩展用况的行为的执行。

见：**扩展关系**(3.1.81)，**包含**(3.1.96)。

3.1.81

扩展关系　extension

一种聚合关系，用来表明通过衍型扩展某个元类的性质，并提供可以方便地对类灵活地增删衍型的能力。

3.1.82

门面　facade

一种衍型化的包，它仅包含对另一个包拥有的模型元素的引用。门面用于为包的某些内容提供“公用视图”。

3.1.83

特征　feature

一种性质，例如某个操作或者属性，它刻画了一个类目的实例。

3.1.84

终[结]状态　final state

一种特殊类型的状态，它表明包含它的组合状态或者整个状态机的完成。

3.1.85

激发　fire

执行某一状态转移，

见：**转移**(3.1.213)。

3.1.86

焦点类　focus class

一种衍型化的类，它定义了提供支持的一个或多个辅助类的核心逻辑或控制流。典型地，焦点类与一个或多个辅助类一起使用。在设计阶段，它对规定构件的核心业务逻辑或控制流特别有用。

见：**辅助类**(3.1.22)。

3.1.87

控制焦点　focus of control

在顺序图中图示一个时段的符号，在该时段内某个对象正在直接地或者通过一个下级过程执行一个动作。

3.1.88

框架　framework

一种衍型包，其中的模型元素为整个系统或者部分系统规约了一个可复用的体系结构。典型地，框架包括类、模式或模板。当框架对某个应用领域做特化时，有时称之为应用框架。

见：**模式**(3.1.142)。

3.1.89

可泛化元素　generalizable element

一种可参与泛化关系的模型元素。

见：**泛化**(3.1.90)。

3.1.90

泛化　generalization

较一般类目与较特殊类目之间的一种分类学关系。每个较特殊类目的实例也是较一般类目的一个间接实例。因此，较特殊类目间接地具有较一般类目的特征。

见：**继承**(3.1.97)。

3.1.91

守卫条件　guard condition

为使相关的转移能够激发而应得到满足的条件。

3.1.92

实现　implementation

对事物如何构造或进行计算的一种定义。例如，类是类型的一种实现，方法是操作的一种实现。

3.1.93

实现类　implementation class

一种衍型化的类，它是按照某种程序设计语言，如 **C++**，**Smalltalk**，**Java**(其中实例或许不能有多个类)的规定对类的实现。如果一个实现类提供了为一个类型定义的所有操作，且具有为该类型的操作指定的同样的行为时，即称该实现类实施了此类型。

见：**类型**(3.1.214)。

3.1.94

实现继承　implementation inheritance

对较一般元素的实现的继承。包括对接口的继承。

相对语：**接口继承**(3.1.104)。

3.1.95

引入　import

在包的语境中，展现对它们的类可以在某一给定包(包括递归地嵌入其内的包)之内加以引用的包的依赖。

相对语：**引出**(3.1.78)。

3.1.96

包含　include

从基用况到包含用况的一种关系，它规定基用况的行为如何容纳包含用况的行为。包含用况的行为包含在基用况中所定义的位置处。基用况依赖于包含用况行为的执行，但不依赖于其结构(即属性或操作)。

见：**扩展**(3.1.80)。

3.1.97

继承　inheritance

较特殊的元素结合较一般元素的结构和行为的机制。

3.1.98

初[始]状态　initial state

一种特殊状态，它表明了到组合状态的默认状态的独立转移的源状态。

3.1.99

实例　instance

具有唯一标识的、能对其应用一组操作并存储这些操作效果的状态的实体。

见：**对象**(3.1.129)。

3.1.100

交互　interaction

为完成某一特定任务，对实例之间如何发送激励的一种规约。交互在协作的语境中定义。

见：**协作**(3.1.39)。

3.1.101

交互图　interaction diagram

适用于侧重对象交互的若干类型的图的一种类属术语。它们包括通信图、顺序图和交互概览图。

3.1.102

交互概览图　interaction overview diagram

以一种活动图的变种来描述交互的图，这种方法关注并提高了对控制流的概览，其中控制流的每个结点都可以是一个交互图。

3.1.103

接口　interface

一个刻画某个元素的行为的命名的操作集合。

3.1.104

接口继承　interface inheritance

对某一较一般元素的接口的继承。不包括对实现的继承。

相对语：**实现继承**(3.1.94)。

3.1.105

内部转移　internal transition

表明不改变对象的状态而对事件做出响应的一种转移。

3.1.106

层　layer

在同一抽象级上对类目或包进行的组织。层表示对体系结构的横向切片，而分区则表示纵向切片。

相对语：**分区**(3.1.141)。

3.1.107

生存线　lifeline

在交互中代表一个独立参与实体的建模元素。一个生存线仅代表一个交互实体。

3.1.108

链[接]　link

某一对象元组中的一种语义连接。关联的实例。

见：**关联**(3.1.18)。

3.1.109

链[接]端　link end

关联端的实例。

见：**关联端**(3.1.20)。

3.1.110

消息　message

以期活动会随之发生，而对信息从一个实例到另一实例的传送的一种规约。用消息可规定对信号的引发或对操作的调用。

3.1.111

元类　metaclass

一种实例为类的类。典型地元类用来构造元模型。

3.1.112

元元模型　meta-metamodel

一种定义用于表达元模型的语言的模型。元元模型与元模型之间的关系，类同于元模型与模型之间的关系。

3.1.113

元模型　metamodel

一种定义用于表达模型的语言的模型。

3.1.114

元对象　metaobject

在元建模语言中用于所有元实体的一种类属术语。

例如，元类型、元类、元属性和元关联。

3.1.115

方法　method

操作的实现。它规定了与某一操作相关的算法或过程。

3.1.116

模型侧面　model aspect

一种侧重于元模型的特定质量的建模维度。

例如，结构模型侧面侧重于元模型的结构质量。

3.1.117

模型细化　model elaboration

从已公布的模型生成知识库类型的过程。包括对接口和实现的生成，这允许基于被细化的模型实例化和装入知识库（二者要相符）。

3.1.118

模型元素　model element

一种元素，它是一个从被建模的系统中做出的抽象。

相对语：**视图元素**（3.1.226）。

3.1.119

模型库　model library

包含供其他包复用的模型元素的衍型化包。模型库不同于外廓之处在于：模型库不采用衍型和标记定义来扩展元模型。模型库类似于某些程序设计语言中的类库。

3.1.120

建模时（期）　modeling time

涉及在软件开发过程的建模阶段所出现的事物。包括分析时和设计时。

用法注：在讨论对象系统时，区分建模时与运行时的有关事物，常常是很重要的。

见：**分析时**（3.1.15）、**设计时**（3.1.64）。

相对语：**运行时**（3.1.170）。

3.1.121

多重分类　multiple classification

一个实例同时直接到多个类目上的指派。

见：**静态分类**（3.1.185）、**动态分类**（3.1.70）。

3.1.122

多重继承　multiple inheritance

泛化的语义变种，其中一个类型可以有多个超类型。

相对语：**单继承**（3.1.178）。

3.1.123

势域（曾称多重性）　multiplicity

曾称多重性。对集合可取的势的范围的规约。势域可以对关联端、组成类中的部分、重复次数或为

其他目的给出规约。本质上,势域是非负整数的一个(可能无限)子集。

相对语:**势**(3.1.31)。

3.1.124

***n* 元关联　*n*-ary association**

三个或更多类之间的一种关联。该关联的每个实例都是取自各类的一个 ***n*** 元组值。

相对语:**二元关联**(3.1.27)。

3.1.125

名[称]　name

用于标识模型元素的一种字符串。

3.1.126

命名空间　namespace

其中可定义和使用名称的模型部分。命名空间中的每一个名称的意义都是唯一的。

见:**名[称]**(3.1.125)。

3.1.127

结点　node

代表运行时计算资源的一种类目,它通常至少有一个存储器,且至少具备处理能力。运行时对象和构件可以驻留在结点上。

3.1.128

注释　note

附加在一个元素或一个元素集合上的注解。注释没有语义。

相对语:**约束**(3.1.54)。

3.1.129

对象　object

类的一个实例。

见:**类**(3.1.34)、**实例**(3.1.99)。

3.1.130

对象图　object diagram

由在某一时刻的对象以及其间关系所组成的一种图。对象图可视作类图或通信图的特例。

见:**类图**(3.1.37),**通信图**(3.1.41)。

3.1.131

对象流状态　object flow state

活动图中的状态,它表示把某个对象从一个状态中的动作的输出传递到另一个状态中的动作的输入。

3.1.132

对象生存线　object lifeline

在顺序图中,一种表示某一对象存在时期的线段。

见:**顺序图**(3.1.175)。

3.1.133

操作　operation

一种特征,它声明了可由类目的实例执行的一个服务。

3.1.134

包　package

一种将元素成组的通用机制。包可以嵌套在其他包内。

3.1.135

包图　package diagram

一种图，它描绘了模型元素如何组织成包以及各包之间的依赖，包括包的引入和包的扩展。

3.1.136

参数　parameter

形[式]参[数]　formal parameter

一个行为特征的实参。一个参数规定了传入和传出一个行为元素（如一个操作）调用的实参。一个参数的类型限制了可以传送的值。

相对语：**变元**（3.1.16）。

3.1.137

参数化元素　parameterized element

模板　template

带有一个或多个未绑定参数的类的描述符。

3.1.138

父　parent

在泛化关系中，对另一元素（即子）的泛化。

见：**子类**（3.1.193）、**子类型**（3.1.198）。

相对语：**子**（3.1.32）。

3.1.139

部件　part

代表一组实例的元素，这些实例被一个包容类目实例或者类目的角色所拥有。

部件可以通过附着的连接件来连接在一起，并且指定在包容类目实例内创建的链实例的配置。

见：**角色**（3.1.169）。

3.1.140

参与　participate

一个模型元素与一个关系或者一个具体化的关系的联系。

例如，类在关联中的参与，参与者在用况中的参与。

3.1.141

分区　partition

基于一组标准对于任何模型元素集合的分组。

a）　活动图：活动结点和边的分组。分区划分结点和边，以约束和显示所包含结点的视图。分区可以共享内容。各分区经常和业务模型中的组织单元对应。也经常把它们用于在一个活动内的结点之间分配特性和资源。

b）　体系结构：一种在同一抽象级或跨分层的体系结构各层的有关类目或包的集合。分区表示对体系结构的纵切片，层则表示横切片。

相对语：**层**（3.1.106）。

3.1.142

模式　pattern

描述了一个设计模式的结构的模板协作。**UML** 的模式要比那些设计模式团体使用的模式受到更多限制。通常，设计模式包含更多的非结构化的方面，比如它们使用的启发和用法的折中。

3.1.143

持久对象　persistent object

一种在创建它的进程或线程不复存在后依然存在的对象。

3.1.144

栓　pin

一种表示数据值的模型元素，这些数据值在行为被调用时传入行为，当行为执行完成时从行为返回。

3.1.145

端口　port

类目的一个特征，它规定了一个在类目与其环境之间或者在类目(的行为)与其内部部件之间的特定交互点。端口通过连接件来连接到其他端口上，通过连接件可以请求对类目的行为特征的调用。

3.1.146

后置条件　postcondition

一种表达约束的条件，在某一操作完成之时条件应为真。

3.1.147

幂类型　powertype

一种其实例又是另一个类目的子类的类目。因此幂类型是一种特别交织的元类：实例也是子类。

3.1.148

前置条件　precondition

一种表达约束的条件，在调用操作时条件应为真。

3.1.149

原子类型　primitive type

一种预定义的、没有任何相关子结构(即不可分解)的数据类型，例如整型或字符串。它还可能具有在 UML 定义之外的代数和操作，例如，数学上的。

3.1.150

规程　procedure

过程

可以作为一个单元附着到模型部件上的动作集合，例如，方法体。因此当一个过程执行时，按照过程参数的规约，获得一组数值作为实参，产生一组数值作为结果。

3.1.151

进程　process

过程

a) 在一个操作系统中的一种重量级的并发和执行单元。操作系统包括重量级与轻量级两种进程。如有必要，可以用衍型对实现加以区别。

b) 一种软件开发过程——用于开发系统的步骤和指南。

c) 执行某一算法，或是做动态处理。

相对语：**线程**(3.1.206)。

3.1.152

外廓　profile

一种衍型包，其内含有为特定领域或目的采用扩展机制加以定制的模型元素，例如衍型、标记定义和约束。外廓还可规定它所依赖的模型库和它所扩展的元模型子集。

3.1.153

投影　projection

从集合到其子集的一种映射。

3.1.154

性质　property

表征元素某一特性的命名值。性质有语义影响。在 **UML** 中预定义了一些性质;另一些可由用户定义。

见:**标记值**(3.1.204)。

3.1.155

伪状态　pseudo-state

在状态机中,一种虽有状态形式却无状态行为的顶点。伪状态包括初始顶点和历史顶点。

3.1.156

物理系统　physical system

a)　模型的主体。

b)　连接起来的物理单元的集合,包括软件、硬件和人,它们被组织起来实现一个特定的目的。一个物理系统可以被描述为一个或多个模型,可能从不同的视点加以描述。

相对语:**系统**(3.1.203)。

3.1.157

限定符　qualifier

一种关联属性或属性元组,其值将一组对象加以划分,这组对象与某一关联另一端处的一个对象相关。

3.1.158

实施　realization

两个模型元素集合之间特化的抽象关系,其中一个集合表示规约(提供者),另一集合表示前者的实现(客户)。实现可用于对模型进行逐步精化、最优化、转换、模板、模型合成或框架组合,等等。

3.1.159

接收(消息)　receive　(a message)

对来自发送对象的激发的处理。

见:**发送方**(3.1.174)、**接收方**(3.1.160)。

3.1.160

接收方　receiver

处理由发送对象传递的激发的对象。

相对语:**发送方**(3.1.174)。

3.1.161

接收　reception

类目准备对收到信号做出反应的声明。

3.1.162

参考　reference

a)　模型元素的指称。

b)　类目中命名的槽,它可以简化到其他类目的导航。

同义语:**参数**(3.1.136)。

3.1.163

精化　refinement

对已经在某一详细程度上描述的事物的更详细描述的关系。

例如:设计类是对分析类的精化。

3.1.164

关系　relationship

描述元素之间某种连接的抽象概念。关系的例子有关联和泛化。

3.1.165

储存库　repository

用来存储对象模型、接口及实现的设施。

3.1.166

需求　requirement

系统所需要的特征、性质或行为。

3.1.167

责任　responsibility

类目的契约或职责。

3.1.168

重用　reuse

对已经存在制品的使用。

3.1.169

角色　role

在参与特定语境的实体集合上定义的特征的命名集合。

协作:由类或参加某种特定语境的部件所拥有的行为的命名集合。

关联:与关联端的同义语,通常引用参与关联中的类目实例的子集。

3.1.170

运行时[期]　run time

计算机程序或系统执行的一段时间。

相对语:**建模时**(3.1.120)。

3.1.171

场景　scenario

说明行为的动作的特定序列。场景可用来说明交互或用况实例的执行。

见:**交互**(3.1.100)。

3.1.172

语义变化点　semantic variation point

元模型的语义的变化点。它为解释元模型的语义提供一种有意的自由度。

3.1.173

发送(消息)　send　(a message)

从发送实例到接收实例的激发传递。

见:**发送方**(3.1.174)、**接收方**(3.1.160)。

3.1.174

发送方　sender

向接收方实例传递激发的对象。

相对语:**接收方**(3.1.160)。

3.1.175

顺序图　sequence diagram

描绘交互的图,侧重于随着在生命线上消息对应的事件的发生所交换的消息顺序。

与通信图不同的是，顺序图包括了时间序列但是不包括对象关系。顺序图既能以一种一般方式（描述所有可能的场景）存在，也能以一种实例的方式（描述一个实际的场景）存在。顺序图和通信图表达了相似的信息，但方式不同。

见：**通信图**（3.1.41）。

3.1.176

信号　signal

对异步的激发的规约，异步的激发以异步的方式激起接收方的反应，而无需回复。接收对象根据其接收的规定处理该信号。由发送请求承载的数据，以及通过引起发送请求的发送调用事件的发生传递给它的数据，由信号实例的属性来表示。信号的定义独立于处理该信号的类目。

3.1.177

特征标记　signature

行为特征的名称和参数。特征标记可以包括一个可选的返回参数。

3.1.178

单[一]继承　single inheritance

多继承　multiple inheritance

泛化的语义变种，其中一个类型只有一个父类型。

相对语：**多重继承**（3.1.122）。

3.1.179

槽　slot

一种规约，用来描述由实例的规约建模的实体，它具有针对特定的结构特征的值。

3.1.180

软件模块　software module

软件存储和操作的单元。软件模块包括源代码模块、二进制代码模块和可执行代码模块。

3.1.181

规约（规格说明）　speciafication

系统或其他类目的需求集合。

相对语：**实现**（3.1.92）。

3.1.182

状态　state

对象在其生存周期中满足某一条件、进行某种活动或等待某一事件的条件或状况。

3.1.183

状态机图　state machine diagram

描绘通过有限状态转换系统建模的离散行为的图。特别是，它描述对象或交互在其生存期间对事件响应及其响应和动作的状态序列。

见：**状态机**（3.1.184）。

3.1.184

状态机　state machine

描述对象或交互在其生存期间对事件响应及其响应和动作的状态序列的行为。

3.1.185

静态分类　static classification

由实例到类目的指派，这一指派不能再更改为其他的类目。

相对语：**动态分类**(3.1.70)。

3.1.186

衍型　stereotype

一个类，它定义了现有的元类(或衍型)可以如何扩展，并使得特定平台或领域的术语或表示法，以及其他东西可用于被扩展的元类。**UML** 预定义了一些衍型，其他的可由用户定义。衍型是 **UML** 的一种扩展机制。

见：**约束**(3.1.54)、**标记值**(3.1.204)。

3.1.187

激励　stimulus

信息从一个实例到另一实例的传递，例如发出一个信号或调用一个操作。信号的接收通常认为是一个事件。

见：**消息**(3.1.110)。

3.1.188

串　string

文本字符的序列。串的表示细节依赖于实现，可包括支持国际化字符和图形的字符集。

3.1.189

结构特征　stuctuaral feature

模型元素的静态特征，例如属性。

3.1.190

结构模型侧面　structural model aspect

强调系统中对象的结构的模型方面，包括其类型、类、关系、属性和操作。

3.1.191

结构图　structure diagram

一种描绘规约中与时间无关的元素的图的形式。类图和构件图都是结构图的例子。

3.1.192

子活动状态　subactivity state

活动图中的状态，用来表示具有一定持续时间的非原子的步骤序列的执行。

3.1.193

子类　subclass

在泛化关系中，对另一类——父类的特化。

见：**泛化**(3.1.90)。

相对语：**父类**(3.1.199)。

3.1.194

子机状态　submachine state

状态机中等价于组合状态，但其里面的内容由另外的状态机描述的状态。

3.1.195

子状态　substate

作为组合状态一部分的状态。

见：**并发子状态**(3.1.51)、**不相交子状态**(3.1.67)。

3.1.196

子包　subpackage

包含在另一个包中的包。

3.1.197

子系统　subsystem

大型系统的层次分解单元。子系统通常间接的实例化。子系统的定义因领域和方法的不同而差异很大,并且希望领域和方法外廓能对该构造进行特化。子系统可定义为具有规约元素和实现元素。

3.1.198

子类型　subtype

在泛化关系中,对另一类型——父类型的特化。

见:**泛化**(3.1.90)。

相对语:**父类型**(3.1.200)。

3.1.199

父类　superclass

在泛化关系中,对另一类——子类的泛化。

见:**泛化**(3.1.90)。

相对语:**子类**(3.1.193)。

3.1.200

父类型　supertype

在泛化关系中,对另一类型——子类型的泛化。

见:**泛化**(3.1.90)。

相对语:**子类型**(3.1.198)。

3.1.201

供方　supplier

提供可由其他类目调用的服务的类目。

相对语:**客户**(3.1.38)。

3.1.202

同步状态　synch state

状态机中用来同步状态机的并发区域的顶点。

3.1.203

系统　system

有组织的一组元素,每个元素在功能上作为一个单元,也是模型中的顶层子系统。

3.1.204

标记值　tagged value

以"名称—值"对的形式对性质的精确定义。在标记值中,名称称为标记。**UML** 中预定义了一些标记;其他的可以用户自定义。标记值是 **UML** 的三种扩展机制之一。

见:**约束**(3.1.54)、**衍型**(3.1.186)。

3.1.205

模板　template

同义语:**参数化元素**(3.1.137)。

3.1.206

(控制)线程　thread　(of control)

遍历程序、动态模型或控制流的其他某种表示的单个的执行路径。也是用于主动对象的实现(作为轻量进程)的衍型。

见：**进程**(3.1.151)。

3.1.207

计时事件　time event

表示自进入当前状态后所经过的时间的事件。

见：**事件**(3.1.74)。

3.1.208

时间表达式　time expression

求值为时间的绝对或相对值的表达式。

3.1.209

时序图　timing diagram

一种交互图，显示在线性时间上生命线(表示类目实例或类目角色)的状态或条件的变化。大多数常见的用法是在显示对象在响应所接受的事件或激发时，对象状态随时间的变化。

3.1.210

顶层　top level

在包容层次结构中指示最上层包的衍型。顶层衍型定义了寻找名字的外部限制，正如名称空间向外“看”到的那样。例如，子系统 opLevel 表示子系统包容层次的顶端。

3.1.211

踪迹　trace

表明两个元素之间的历史的或进化关系的依赖，这两个元素表示同一概念，但没有从一个导出另一个的特定规则。

3.1.212

暂时对象　transient object

只在产生它的进程或线程的执行期间存在的对象。

3.1.213

转移　transition

两个状态之间的如下关系：当特定的事件发生并且满足特定的条件时，处于第一个状态的对象将执行某些特定的动作并进入第二个状态。一旦有这样的状态变化，就说该转移被激发了。

3.1.214

类型　type

是一种衍型化的类，它描述对象域以及可应用到这些对象的操作，而不定义这些对象的物理实现。类型不可以包含任何方法，保持它自己的控制线程或是嵌套的也不行。然而，它可以具有属性和关联。尽管对象最多可以具有一个实现类，但它可以符合多个不同的类型。

见：**实现类**(3.1.93)。

相对语：**接口**(3.1.103)。

3.1.215

类型表达式　type expression

求值为一个或多个类型的引用的表达式。

3.1.216

未解释　uninterpreted

对其实现没有在 **UML** 中描述的类型留下的占位符。每一个未解释的值都有一个相应的字符串表示。

3.1.217

使用　usage

一种依赖关系，其中的一个元素（客户）为了正确地运行或实现，要求另一元素（提供方）的存在。

3.1.218

用况　use case

对系统（或其他实体）要完成的动作序列（包括变体）的规约，它们与系统的参与者交互。

见：**用况实例**（3.1.220）。

3.1.219

用况图　use case diagram

描绘参与者、主题（系统）、用况之间关系的图。

3.1.220

用况实例　use case instance

用况中所描述的动作序列的执行。用况的实例。

见：**用况**（3.1.218）。

3.1.221

用况模型　use case model

利用用况描述系统的功能需求的模型。

3.1.222

公用设施　utility

一种用类的声明的形式将全局变量和过程集合起来的衍型。它的属性和操作分别变为全局变量和全局过程。公用设施不是一种基础的建模构造物，是为了方便编程而设立的。

3.1.223

值　value

类型域中的元素。

3.1.224

顶点　vertex

状态机中转移的源或目标。顶点可以是一个状态或伪状态。

见：**状态**（3.1.182）、**伪状态**（3.1.155）。

3.1.225

视图　view

从一个特定的视角或有利位置看到的模型的投影，并且忽略与该视角不相关的实体。

3.1.226

视图元素　view element

一组模型元素的文本或（和）图形的投影。

3.1.227

视图投影　view projection

模型元素到视图元素的投影。视图投影为每个视图元素提供位置和样式。

3.1.228

可见性　visibility

一种枚举，其值（公用的，受保护的，或私用的）用来指明它所针对的模型元素如何在封装它的命名空间之外被看到。

3.2 缩略语

CWM	公共仓库元模型(Common Warehouse Metamodel)
MOF	元对象设施(Meta-Object Facility)
OMG	对象管理组织(Object Management Group)
RTF	UML 修订任务组(Revision Task Forces)
RFP	提案请求(Request for Proposal)
UML	统一建模语言(Unified Modeling Language)
XMI	XML 元数据交换(XML Metadata Interchange)

4 语言体系结构

4.1 综述

本部分是通过采用形式规约技术的元建模方法定义的(也就是用元模型来规定组成 UML 的模型)。虽然这种方法缺乏形式化规格说明方法应具备的一些严密性,但对大多数专业技术人员而言,它具有更强的直观性和实用性。本章说明 UML 元模型的体系结构。

以下各条概括了所遵循的设计原则,展示了如何运用这些原则来组织 UML 的基础结构和上层结构。最后说明了 UML 元模型如何与四层结构的元模型体系结构模式取得一致。

4.2 设计原则

UML 元模型的体系结构设计遵循以下的设计原则:

a) 模块性。这一原则强调高内聚性和低耦合性,适用于将各构造分组为若干包,并将各特征组织到元类之中。

b) 分层。分层原则以两种方式运用于 UML 元模型。

   1) 将包结构分层,以使元语言核心构造与使用它们的高级构造分离开来;

   2) 四层元模型体系结构模式一致地适用于分离各抽象层的内容(尤其是实例化时)。

c) 划分。划分原则用于组织同一层的概念域。对于基础结构库,采用细粒度的划分,以提供当前和将来元建模标准所需的灵活性。对 UML 元模型采用粗粒度划分,以增强包的内聚力并减弱包之间的耦合性。

d) 可扩展性。UML 能以两种方式扩展:

   1) 采用外廓定义新的 UML 方言,以便为特殊的平台(比如 J2EE/EJB 和. NET/COM+)和领域(比如金融、电信和航天)定制语言;

   2) 可以通过重用基础结构库包中的部分内容,并增加适当的元类和元联系来规定与 UML 相关的新语言。前者定义了 UML 的新方言,后者定义了 UML 语言系列中的新成员。

e) 重用。提供一种细粒度的灵活的元模型库,用以定义 UML 元模型以及在体系结构上有关的其他元模型,比如,元对象设施(MOF)和公共仓库模型(CWM)。

4.3 基础结构的体系结构

UML 的基础结构是由基础结构库来定义的。基础结构库满足以下设计要求:

a) 定义元语言核心,该核心可以重用以定义各种元模型,包括 UML、MOF 和 CWM。

b) 在体系结构上使 UML、MOF 和 XMI 衔接,这样模型互换得到充分支持。

c) 允许通过外廓定制 UML,并基于与 UML 相同的元语言核心来创建新语言(语言系列)。

如图 1 所示,基础结构由 InfrastructureLibrary 包表示。基础结构库包由核心包(Core)和外廓包(Profile)组成,前者包括了建立元模型时所用的核心概念,后者定义了定制元模型的机制。

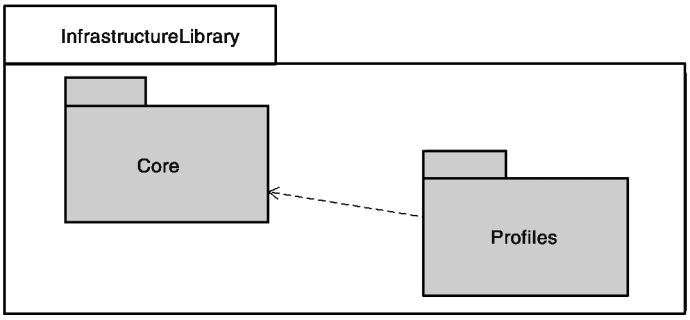


图 1　基础结构库包

4.3.1　核心(Core)

核心包的首要作用是为了支持高重用性而特别设计的完整的元模型,而处于相同元层的其他元模型(见 4.3.4)或者引入或者特化其特定的元类。UML、CWM 和 MOF 如何依赖于共有核心,如图 2 所示。由于这些元模型正好处于模型驱动的体系结构(MDA)的中心,所以此公共核心也可认为是 MDA 的体系结构内核。其目的是使 UML 和其他 MDA 元模型重用该核心包的部分或全部,这使其他元模型可利用已经定义的抽象语法和语义。

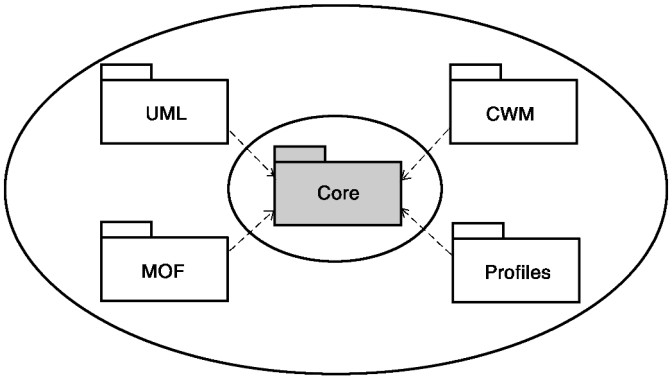


图 2　公共核心包的作用

为了便于重用,将核心包细分为若干子包:原子类型包(Primitive Type)、抽象包(Abstraction)、基本包(Basic)和构造包(Construct),如图 3 所示。

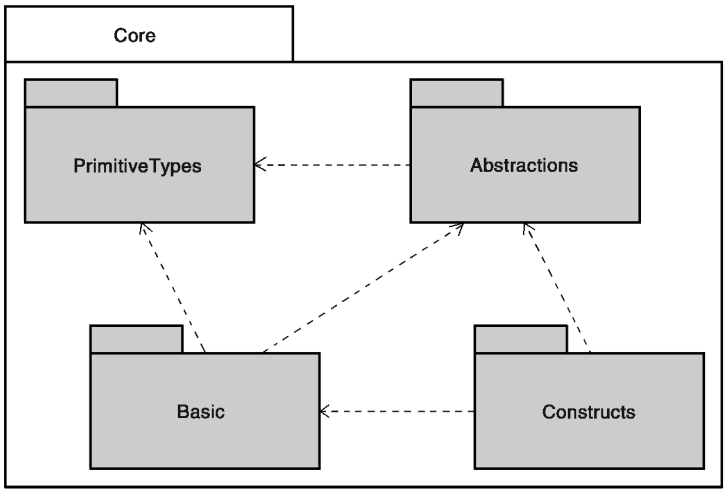


图 3　核心包

在接下来的章条中将看到，其中部分子包进一步划分为粒度更细的包，从而在定义一个新的元模型时，可以挑选有关的部分。不过应该注意：选择特定的包意味着同时选择了依赖的包。原子类型包只包含了少数已定义的在建元模型时常用的类型，是为满足 UML 和 MOF 的需要而专门设计的。其他元模型可能需要其他的或相重叠的原子类型集。其余两个包的设计原理差别不大。抽象包大多包括抽象元类，用来进一步特化或由很多元模型重用。对于想重用这种包的元模型很少做出假设，为此，也把抽象包分为若干更小的包。另一方面，构造包大多包括具体元类，主要用于面向对象的建模。构造包主要由 MOF 和 UML 两者重用，并且成为衔接这两个元模型的重要部分。基本包表示几种构造，用作 UML、MOF 和其他基于基础结构库的元模型所产生的 XMI 的基础。

核心包的第二个作用是用来定义创建元模型所用的建模构造。这通过对基础结构库（见“元模型的分层”）中的元类进行实例化来实现。通过 MOF 对元类进行实例化，而基础结构库定义实际的元类。用来实例化 UML、MOF 和 CWM 中的元素，实际上是基础结构库自身的元素。从这个角度考虑，称基础结构库是自我描述的或自反的。

4.3.2 外廓（Profiles）

如图 1 所示，外廓包依赖核心包，并定义了用于对现存的元模型进行裁减的机制，以适应特定的平台，比如 C++、CORBA 或 EJB，或者实时、商业对象、软件过程建模等领域。虽然引入外廓的首要目标是针对 UML，但也可以和任何基于公共核心（即从其实例化而来）的元模型一起使用。外廓应基于 UML 等所扩展的元模型上，单独使用意义不大。

外廓已经与 MOF 的扩展机制相衔接，但提供了更便捷的限制方法来保证外廓的实现，并且使用更为直接，更容易得到工具厂商的支持。

4.3.3 UML 和 MOF 之间的体系结构衔接

基础结构的一个主要目标一直是在体系结构上将 UML 和 MOF 衔接。实现这一目标的首选途径是定义公共的核心，这在核心包中已予实现，这样模型元素就可以在 UML 和 MOF 间共享。第二种途径是确保将 MOF 定义为基于用作元模型的 MOF 的模型，如图 4 所示。注意 MOF 不仅仅用作 UML 的元模型，同时也用作 CWM 等其他语言的元模型。

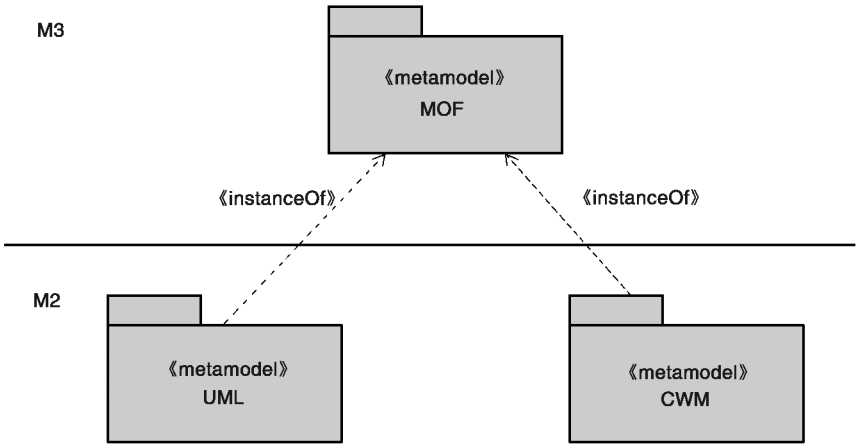


图 4 在不同元层的 UML 和 MOF

各元层间的关系将在“4.3.4”中详细说明。这里值得指出的是，UML 的每一模型元素都是 MOF 中恰好一个模型元素的实例。注意，基础结构库 InfrastructureLibrary 同时在 M2 和 M3 两个元层上使用，这是由于分别由 UML 和 MOF 重用的缘故，如图 2 所示。在 MOF 的情况，InfrastructureLibrary 中的元类按其本义使用，而在 UML 的情况，这些模型元素则增加了一些性质。产生这种差异的原因

是，当对多样性的应用建模时，其要求和建元模型时的要求总有细微的不同。

例如，MOF 定义了如何使用 XML 元数据交换（XMI），使得 UML 模型在工具间相互转换。MOF 不仅为其自身，而且为 CWM、UML 和其他作为 MOF 实例的元模型的内省机制定义了自反接口（MOF::Reflection）。它更定义了作为外廓替代或与外廓联合来扩展元模型的机制（见第 11 章）。事实上，外廓定义为 MOF 扩展机制的一个子集。

4.3.4 上层结构的体系结构

UML 上层结构元模型由 UML 包来规定，其中 UML 包分成处理结构性和行为性建模的包，如图 5 所示。

上述每个部分在 GB/T 28174.2 的各章分别叙述。注意，有些包之间相互依赖，形成循环依赖性。这是由于顶层包之间的依赖性概括了其子包之间的所有联系；这些包的子包之间没有循环依赖性。

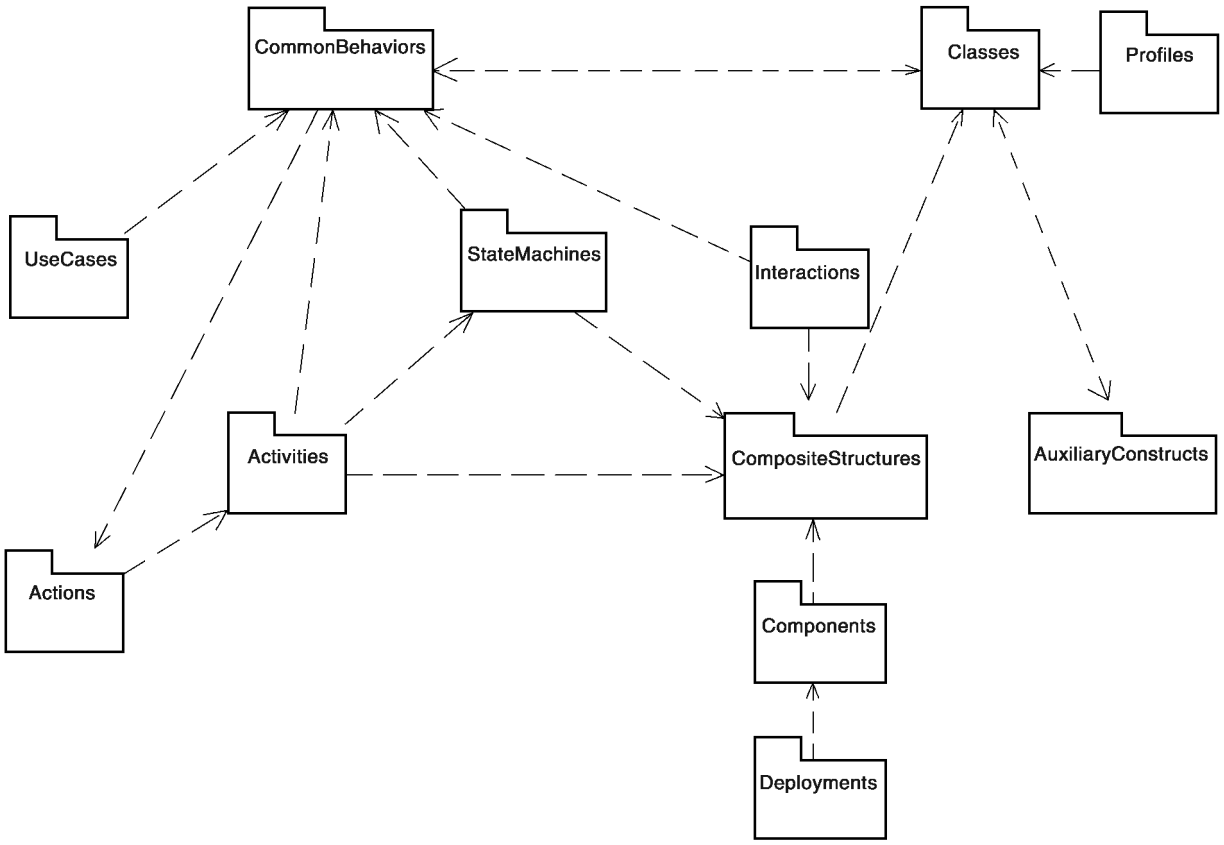


图 5 GB/T 28174.2 上层结构的顶层包结构

4.3.5 重用基础结构

本部分的基础结构规格说明的主要用途之一就是在创建其他元模型时进行重用。UML 元模型重用基础结构库（InfrastructureLibrary）采用两种不同的方式：

UML 元模型全部是 InfrastructureLibrary 中定义的元元类的实例。

UML 元模型引入并特化 InfrastructureLibrary 中的所有元类。

如同前面所讨论的一样，模型可以用作元模型，此处利用这一事实。InfrastructureLibrary 既可用作元元模型，又可用作为元模型，所以从两方面进行重用。

4.3.6 内核包(The Kernel Package)

InfrastructureLibrary 主要在类的内核包中重用,这种重用利用包合并把基础结构中不同包集中起来加以完成。内核包正好处于 UML 的中心,其他包的元类都直接或间接地依赖于它。内核包与 InfrastructureLibrary 中的构造包十分相似,但在建模构造中添加了更多能力,而这对于 MOF 的重用或衔接是不应纳入的。

由于基础结构已经根据重用的思想进行设计,所以在几个不同的包中都有部分地定义了的元类——尤其是抽象包中的元类。大多数情况下,这些不同的包中的元类被集中到构造包中的一个单一的元类,但是某些情况下只有在内核包中这样做。一般来说,当具有相同名称的元类出现在多个包中时,就意味着表示同一元类,而且所定义(特化)的包表示特定的因素分解。在上层结构中也出现这种具有相同模式的部分定义,比如元类 Class 被分解到不同的包中形成一致点(见以下内容)。

4.3.7 元模型层

以核心包为中心的体系结构是四层元模型层次结构的补充视图,传统的 UML 元模型都以这种四层结构为基础。在处理元层以定义语言时,通常需要考虑三层:

a) 语言规格说明或元模型;

b) 用户规格说明或模型;

c) 模型对象。

可以多次递归地应用这种结构,这样就可能得到无限多的元层。在某种情况下的元模型可以是另一情况下的模型,UML 和 MOF 都有这种情况。UML 是一种语言规格说明(元模型),用户可以用以定义自己的模型。类似的,MOF 也是一种语言规格说明(元模型),用户也可用以定义自己的模型。不过从 MOF 的观点来看,UML 是一种基于作为语言规格说明的 MOF 的用户(即 OMG 中已经开发了这种语言的成员)规格说明。在四层元模型的层次结构中,通常将 MOF 称为元元模型,虽然严格来讲它只是一种元模型。

4.3.8 四层元模型的层次结构

元元建模层形成了元模型建模层次结构的基础。这一层的主要任务就是定义规定元模型的语言。这一层通常称为 M3,MOF 是元元模型的一个例子。元元模型通常比它所描述的元模型要简洁,而且定义好几个元模型。一般希望元元模型与有关的元模型共享公共的设计原则和构造。不过,可以认为每一层都独立于其他各层,且需要保持其自身的设计完整性。

元模型是元元模型的实例,这就意味着元模型的每一个元素都是元元模型中的元素的一个实例。元模型层的主要任务就是定义用于规定模型的语言。这一层通常称为 M2、UML 和 OMG 的公共仓库模型是元模型的例子。元模型一般比描述它的元元模型更为详细,特别是在用以定义动态语义时。UML 元模型是 MOF 的一个实例(实际上,每一个 UML 元类都是 InfrastructureLibrary 中一个元素的实例)。

模型是元模型的一个实例。模型层的主要职责是定义描述语义域的语言,也就是说,允许用户对不同领域的问题(比如软件的、商业过程的及需求的)进行建模。被建模的事物都处于元模型层次之外。这一层通常称为 M1。用户模型是 UML 元模型的实例。注意,用户模型既包含了模型元素,又包含这些模型元素的实例的快照(说明)。

M0 层位于元模型层次的底部,它包含了在模型中定义的模型元素在运行时的实例。在 M1 层中建模的快照,是 M0 运行时的受约束情形。

当处理到多于三层的元层时,通常会遇到这种情况:位于 M2 以上的各层,顺着这个层次越往上结构越小越紧凑。对 MOF,位于 M3 层,因而仅共享在 UML 中定义的元模型的一部分。元模型建模的一个特定特征是将语言定义为具有自反性(reflective)(即能定义自身的语言)的能力。InfrastructureLibrary 就是这样的一个例子,因为它包含了需要定义自身的所有元类。当一个语言具有自反性时,就不需要去定义另外一种语言来规定其语义。MOF 由于是基于 InfrastructureLibrary 的,

因此具有自反性，从而不需要在 MOF 之上添加一个元层。

4.3.9 元模型建模

进行元模型建模时，重点是把元模型和模型区分开来。正如上面所述，从元模型实例化而来的模型，可通过递归方式反过来成为另一模型的元模型。一个模型通常包含模型元素。这些元素是通过实例化元模型的模型元素（即元模型元素）来创建。

元模型的一个主要作用是定义模型中的模型元素如何实例化的语义。考虑图 6 中的例子，图中，元类 Association 和 Class 都定义为 UML 元模型的一部分。这些元类（Association 和 Class）在用户模型中以如下方式加以实例化：类 Person 和类 Car 都是元类 Class 的实例，在类 Person 和类 Car 之间的关联 Person. car 是元类 Association 的一个实例。UML 的语义定义了当用户定义的模型元素在 M0 层被实例化时，会发生些什么，并且得到：Person 的一个实例、Car 的一个实例，及这两个实例之间的一种链接，即关联的实例，见图 6。

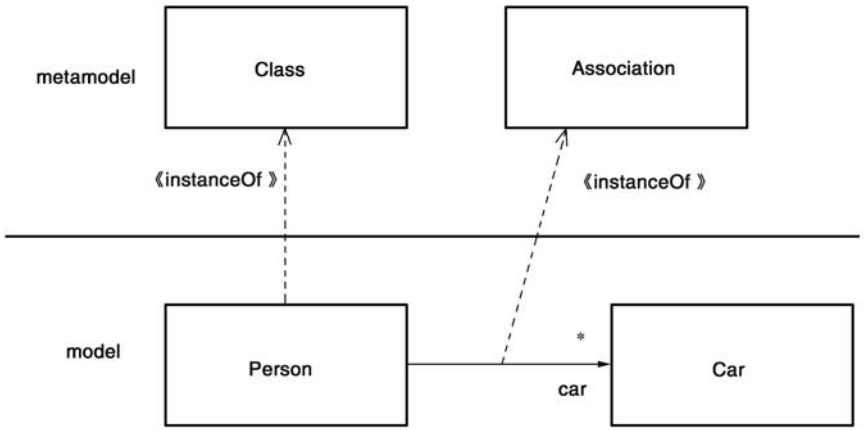


图 6 元模型建模的实例（注意并未图示出所有联系的实例）

这些在 M0 层生成的实例，比如上面的 Person，有时称为“运行时”实例，不应将它与同样被定义为 UML 元模型的一部分的元类 InstanceSpecification 的实例相混淆。InstanceSpecification 的实例，在同一层，在模型中定义为它所说明的模型元素，正如图 7 所示，图中实例规格说明 Mike 是类 Person 的一个实例的说明（或快照）。

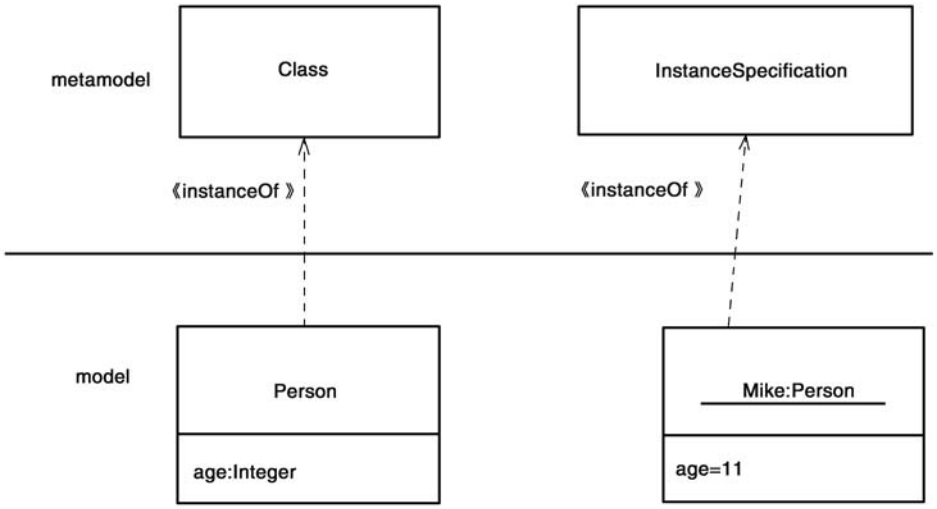


图 7 对实例规格说明的说明

4.3.10 四层元模型层次的例子

图 8 中说明了这些元层之间的关系。应当注意，绝不是仅仅局限于这里的四层元层，有可能添加附加的层。正如图中所示，这些元层从 M0 开始往上递增编号，这取决于究竟使用了多少层。在这一特别情形，编号达到 M3 时，对应于 MOF。

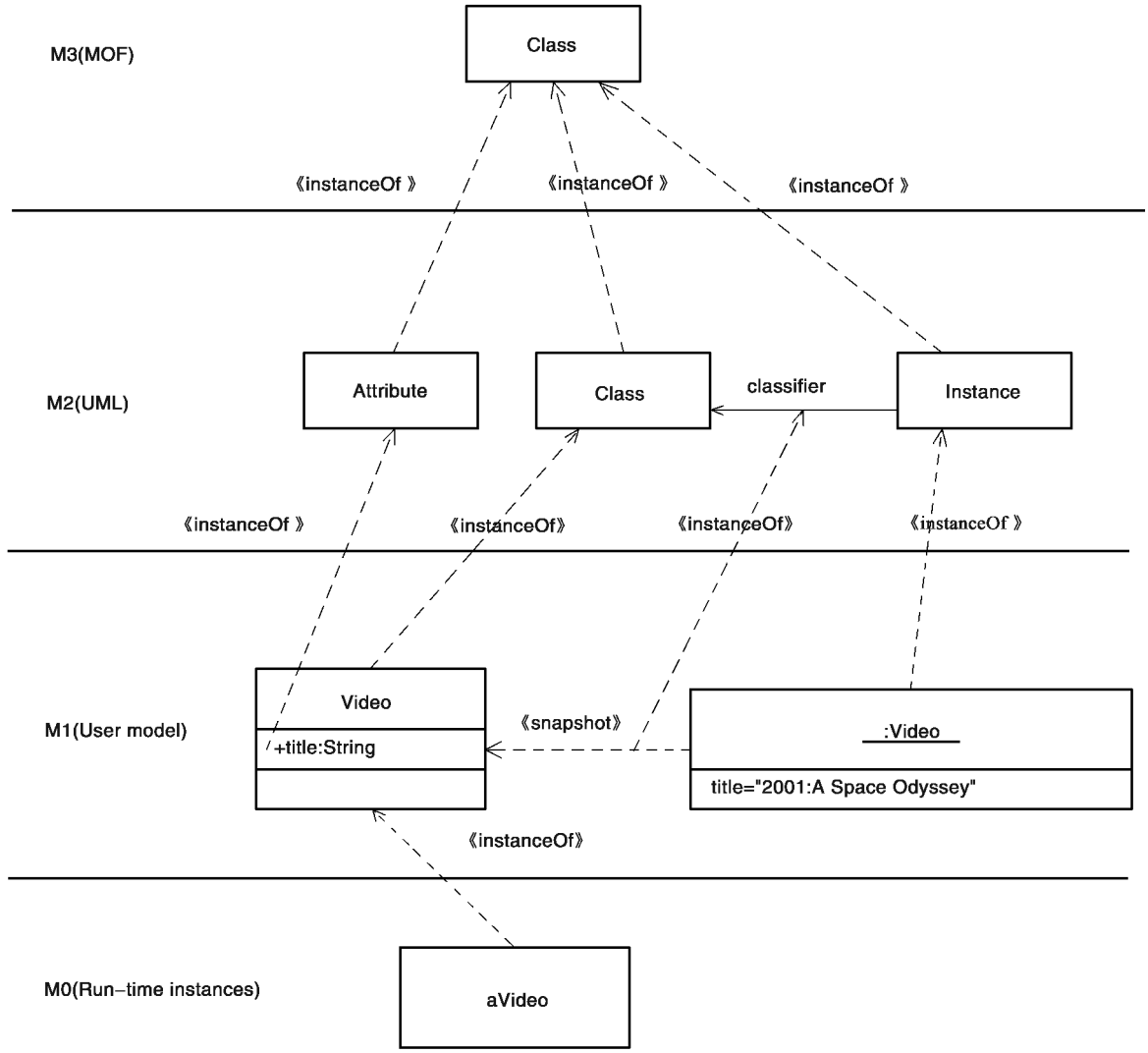


图 8 四层元模型层次结构的实例

5 语言形式体系

5.0 概述

UML 规格说明是按元建模方法定义，这种方法采用了形式规格说明技术。形式规格说明技术用以提高规格说明的精确度和正确性。本章阐述定义 UML 所用的规格说明技术。

使用规格说明技术定义 UML 有如下目标：

a) 正确性——规格说明技术通过帮助进行确认来改进元模型的正确性，例如，良构性规则应有助于确认抽象语法的有效性和辨识差错。

b) 精确性——规格说明技术应同时提高语法和语义两方面的准确性。应保证足够的准确性使得对实现方和用户来说，在语法上和语义上都没有歧义。

c) 简明性——规格说明技术应是高度精练的，以便准确地定义语法和语义而免除多余的细节。

d) 一致性——规格说明技术通过以一致的方式添加实质性细节来补充元建模方法。

e) 可理解性——在提高精确性和简明性的同时，规格说明技术也应该改进规格说明的可读性。因此，由于严格的形式体系，UML 规格说明中较少使用严格的形式体系。

规格说明技术同时运用文字和图形两种表示以三种视图来描述元模型。

需要强调的是，当前的描述并不是完全形式化的语言规格说明，因为那样做会大大增加复杂度而不会带来明显的好处。

虽然如此，仍对语言的结构赋予了精确的规格说明，这是工具互操作性的需要。详细语义采用自然语言以精确方式描述，使其容易理解。当前看法是，语义对工具开发并不是本质的；不过这种情况将来有可能改变。

5.1 形式化方法的等级

语言规格说明的常用技术是先定义语言的语法，后描述其静态和动态语义。语法定义语言中存在什么构造，这些构造又如何借助其他构造来构建。有时，尤其是当语言含有图形语法时，用一种与记法无关的方式定义语法（即定义该语言的抽象语法）非常重要。然后，通过将记法映射到抽象语法上来定义具体语法。

语言的静态语义定义构造的实例应如何与其他实例连接才有意义，动态语义则规定良构性构造的意义。只有当用这种语言写成的描述是良构性时（即当它满足在静态语义中定义的规则时），这种描述的意义才加以定义。

规格说明综合采用了三种语言——UML 的一个子集、一种对象约束语言及精确的自然语言来描述整个 UML 的抽象语法和语义。这种描述是自足式的，阅读本文档不需要参考其他资料来源。虽然这是一种元循环的描述，但因只需对 UML 构造中很小的子集来描述它的语义，所以理解这个文档是实际可行的。

在构造 UML 元模型时，一直运用了不同的技术来规定语言构造，包括采用了 UML 本身的一些能力。将主要的语言结构具体化到元模型的元类之中。其他构造，实质上是构造的变体，定义成元模型中元类的衍型。这种机制允许构造的变体在语义上与基本元类有很大的区别。

定义变体的另一种更“轻量级”的方法是利用元属性。例如，聚合构造是由元类关联端的属性规定的，其中关联端用来指明关联是普通聚合、组合聚合或者普通关联。

包规格说明的结构

这部分提供 UML 元模型中每个包和每个类的信息。每个包都有一个或多个下列子部分：

类描述

这部分包含类中规定包内所定义的构造的枚举。这部分的开头是一个或多个描绘该构造（即类和它们的联系）的抽象语法图，并带有一些良构性要求（势域或定序）。然后是按字母顺序的每个类的规格说明（见后文）。

5.1.1 图（Diagrams）

通常，如果特定种类的图表示了包中定义的构造，那么就包含描述这种图的部分。

5.1.2 实例模型(Instance Model)

可给出一个例子来说明如何“入住”被包含类的实例模型。例子中的元素是包含在该包(或引入包)中的类的实例。

5.2 类规格说明的结构

5.2.1 描述

这部分包括规定 UML 中构造的元类的非形式化定义。并陈述该元类是否是抽象的。

这部分连同下面两部分组成对该构造的抽象语法的描述。

属性

这部分列举出了类的每个属性,并带有简短的解释。在这一部分陈述该属性是否是派生的或是否是另一属性的特化。属性势域为 1(UML 中的默认值)时即被抑制。

5.2.2 关联

这部分以同样的方式列出与该类相连的关联的两端。在这一部分陈述该关联是否是派生的或是否是另一关联的特化。如果关联端的势域是“*”,则被抑制(UML 中的默认值)。

当以有向关联取代属性规定时,无向端的势域为“*”(UML 中的默认值),并且不应使用角色名。

5.2.3 约束

元类的良构性规则,除了势域和定序约束在包部分开始的图中定义以外,都定义为元类的不变量集(可能为空);为使此模型有意义,该元类中的所用实例都应满足这些规则。因此这些规则就对元模型中定义的属性和关联规定了约束。大多数不变量由 OCL 表达式以及对该表达式的非形式化解释来定义,但是在有些情况下,不变量由其他方式来表达(在例外情况下使用自然语言)。陈述“无附加约束”指所有良构性规则在超类中与图中表达的势域和类型信息一起表达。

5.2.4 附加操作(可选的)

在很多情况下,OCL 表达式需要对类的附加操作。附加操作作为一个独立的子部分在构造的约束之后进行定义,采用的方法与约束部分相同:非形式解释跟以定义操作的 OCL 表达式。

5.2.5 语义

良构构造的意义采用自然语义定义。

5.2.6 语义变化点(可选的)

在本文档中,术语“语义变化点”始终指 UML 规范中其目的在整个规格说明中已知,但其形式或语义以某种方式变化的部分。语义变化点的目标就是使 UML 的那一部分能在一个特定的情况下或领域内特化。

语义变化点在标准中以下面几种形式出现:

a) 可更改缺省:在这种形式中,标准为语义变化点提供单一的缺省规格说明,但它可能被替换。例如,标准为将状态机的特化提供了一组缺省规则,但这个缺省可以被另一组不同的规则(一般是根据使用了哪个行为的兼容性来进行选择)覆盖(例如在外廓中)。

b) 多重选择:在这种形式中,标准以显式规定若干互斥的选择,其中一个可标注为缺省值。语言设计者可以从中选择其一或定义一个新的;这种类型的变化点的一个例子可在处理状态机的未预期事件中找到;选择包括(a)忽略该事件(默认状态),(b)以显式拒绝该事件,(c)推迟该事件。

c) 未定义的:在这种形式中,标准未提供任何关于语义变化点预定义规格说明。例如,当一个多态操作被启用时,在标准中并没有定义选择所执行方法的规则。

5.2.7 记法

该部分描述构造的记法。

5.2.8 表示选项(可选的)

当对构造有不同的图示方法时,例如,不必对每次出现都图示出该构造的所有部分时,这种可能性在该部分描述。

5.2.9 样式指南(可选的)

样式指南经常使用一个非形式约定来图示一个构造(或其一部分),例如类的名称应以黑体并居中。这些约定在该部分描述。

5.2.10 例子(可选的)

该部分给出一些如何描述构造的例子。

5.2.11 论据(可选的)

分别给出为什么构造及其记法要如此定义的理由。

注:对 UML1.4 的改进,该部分描述与 UML1.4 相比较的改变,以及从 UML1.4 迁移到 UML2.0 的途径。

5.3 约束语言的使用

本规范采用 OCL 来表达良构性规则。为了提高可读性,采用如下约定:

a) Self——作为对定义不变式语境的元类型的引用时可省略,但为清晰起见一直保留。

b) 在迭代汇集的表达式中,尽管在形式上并无必要,但为清晰起见仍使用迭代符。迭代符的类型通常省略,但当增强理解时带上。

c) 在实际使用时“汇集”操作是隐式的。

d) OCL 约束的语境部分不以显式包含,因为它在约束出现的部分是良好定义的。

5.4 自然语言的使用

尽量准确地使用自然语言。例如,描述 UML 语义包括诸如“X 为…提供能力”和“X 是 Y”这样的短语。在各种情况下,都假定是常用的含义,不过对深层的形式描述,即使是这样简短的短语也需要语义规格说明。

下面是一些适用的通则:

a) 当引用某些元类的实例时,常省略“实例”这个词。例如不说“一个类实例”或“一个关联实例”,而只说“一个类”或“一个关联”。加上前缀“a”或“an”就意味着是“…的实例”。同样,说“元素组”就意味着“元类元素的一组(或这组)实例”。

b) 每当一个单词与 UML 中使用的某一构造的名称相重时，指那个构造。

c) 带有前缀“子”、“超”、或“元”之一的术语，合写成一个词（比如：“元模型”和“子类”）。

5.5 约定和版式

在 UML 描述中，一直采用如下惯例：

a) 当引用 UML 中的构造时，采用正式文字而不用它们在元模型中的表示。

b) 以附加名词或形容词来组成元类名时，采用内嵌的大写字母（例如“ModelElement”和“StructuralFeature”）。

c) 元关联的名称与元类名以相同的方式书写（例如“ElementReference”）。

d) 对以附加的名词或形容词组成的名称，采用内嵌大写字母的（小写）首字母（例如“ownedElement”和“all-Contents”）。

e) 布尔元属性名总以“is”开头（例如“isAbstract”）。

f) 枚举类型总以“Kind”结尾（例如“AggregationKind”）。

g) 当在文中引用元类、元关联、元属性等时，总使用它们在该模型中出现的完全相同的名称。

h) 在图中没有可见性属性，因为所有元素都是公有的。

i) 当必选部分不适用于元类时，采用文字“无添加的×××”，其中“×××”是标题名。当可选部分不适用时，即不列入。

6 基础结构库（Infrastructure Library）

这部分描述了 UML 元模型和相关元模型的基础结构库（Infrastructure Library）的结构和内容，如元对象设施（MOF）和公共仓库元模型（CWM）。基础结构库包定义了 UML 的可重用元语言内核与元模型扩展机制。元语言内核能够用于规定各种元模型，包括 UML、MOF 和 CWM。另外，此库中还定义了一种外廓扩展机制，当某些平台和领域不具备完全元建模能力时，利用这种扩展机制可为其定制 UML。基础结构库的顶层包如图 9 所示。

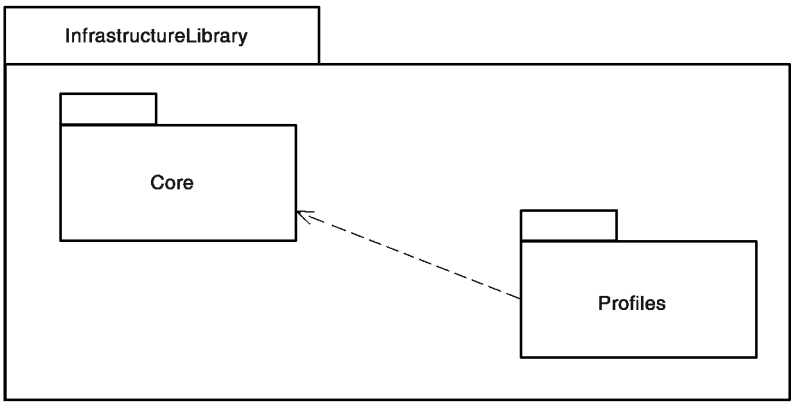


图 9 元模型库包包括核心包和外廓包

核心包是基础结构库中可重用的主体部分，其进一步细分如图 10 所示。

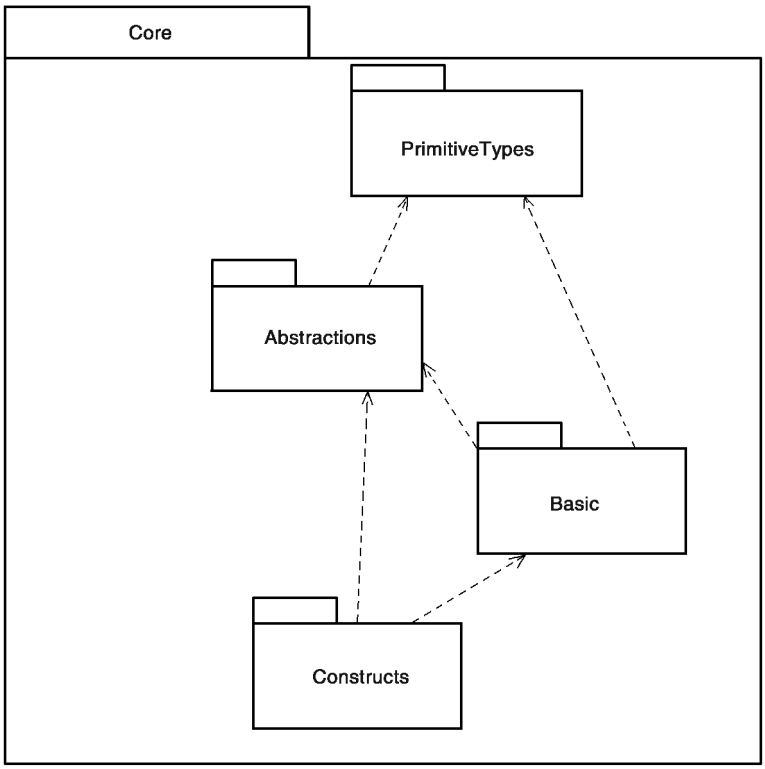


图 10　核心包包括原子类型包、抽象包、基本包和构造包

原子类型(PrimitiveTypes)包比较简单，它包含若干通常用于元模型(metamodeling)建模的预定义类型，因此它们不但用于基础结构库本身，而且用于 MOF 和 UML 等元模型(metamodels)。抽象包包括若干只带有少量元模型的细粒度包，其中大部分是抽象的。这种包的目的是提供高可重用的元类集，在定义元模型时被特化。构造包也包含若干细粒度的包，且把抽象包的多个方面集中在一起。构造包中的元类趋向于具体而不是抽象，并且适用于面向对象建模范型。来看一下元模型(如 MOF 和 UML)，它们通常因为要自动输入核心中其他包的内容而引入构造包。基本(Basic)包包括一个主要为 XMI 所用的构造包的子集。

外廓(profiles)包含用于创建特定元模型(尤其是 UML)外廓的机制。这种扩展机制是 MOF 提供的通用扩展能力的子集。

原子类型包、抽象包、基本包、构造包及外廓包的详细结构和内容将在后续各条进一步描述。

7　核心包::抽象包(Core::Abstractions)

7.0　概述

“基础结构库::核心包”中的抽象包划分为粒度更细的若干子包，这样有利于在创建元模型时灵活地重用。如图 11 所示。

抽象包(Abstractions)的全部子包如图 12 所示。

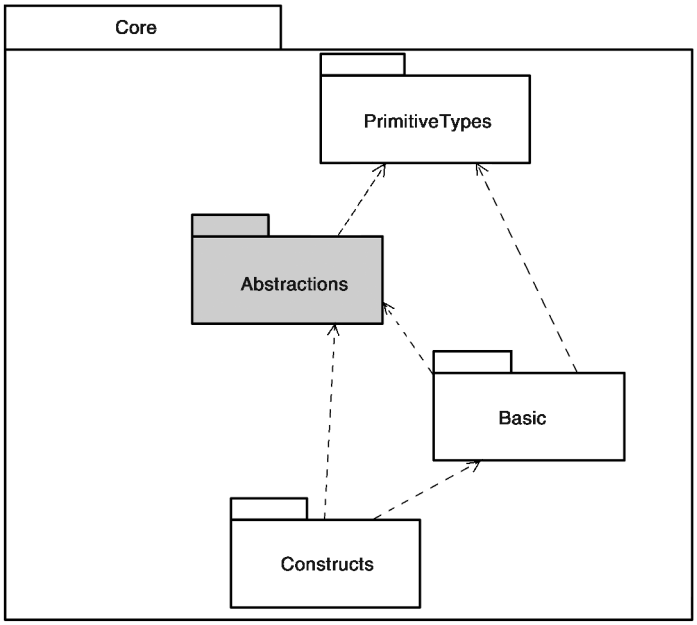


图 11　核心包属于基础结构库包(且包含多个子包)

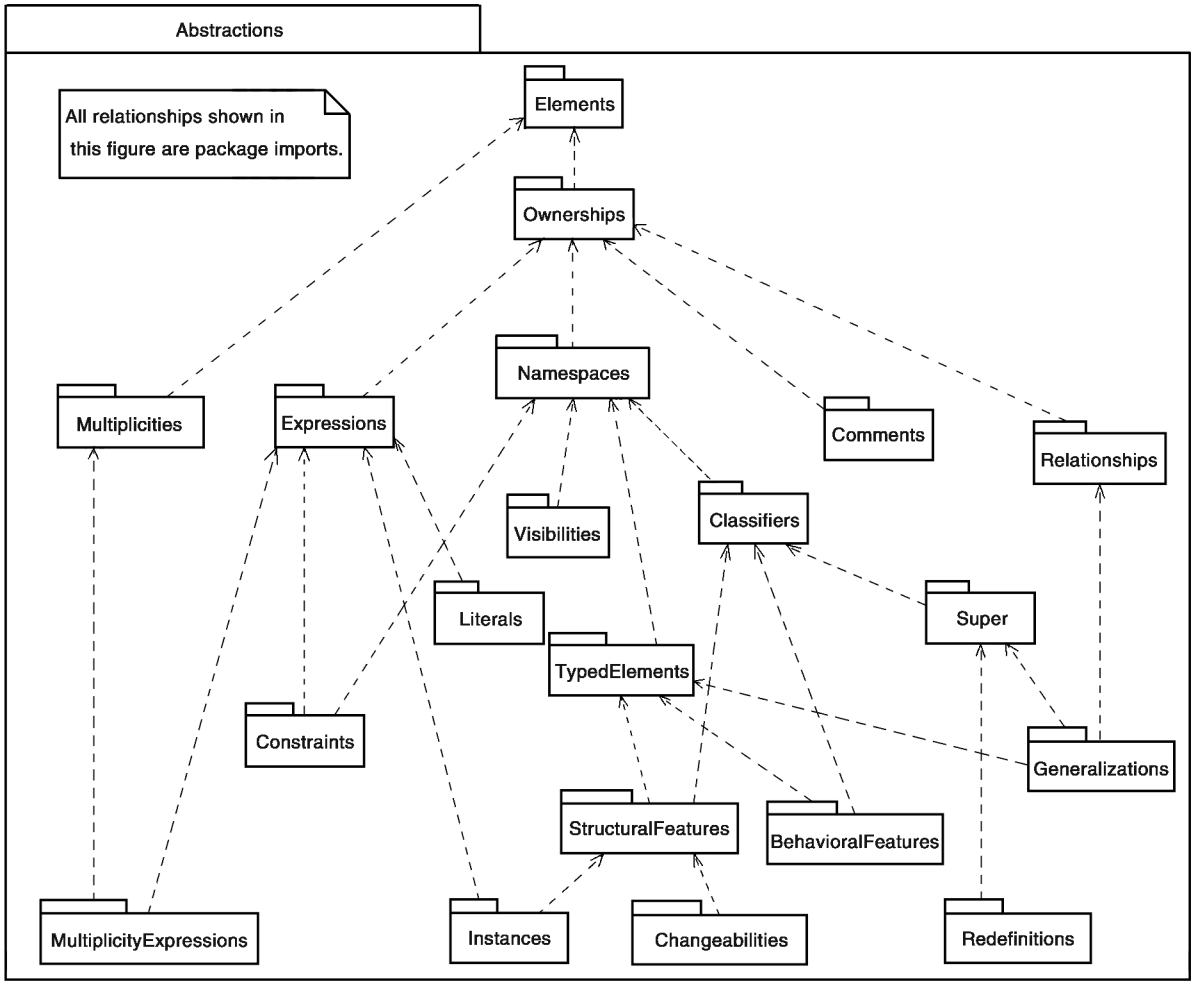


图 12　抽象包包含本章中规定的多个子包

抽象包各子包的内容将在后面分别描述。

7.1 行为特征包(BehavioralFeatures)

抽象包的行为特征包定义了一些基本类,这些基本类用来对模型元素的动态特征进行建模。行为特征包如图 13 所示,行为特征包中定义的元素如图 14 所示。

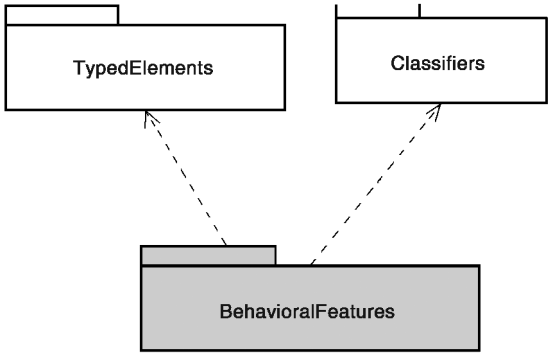


图 13 行为特征包

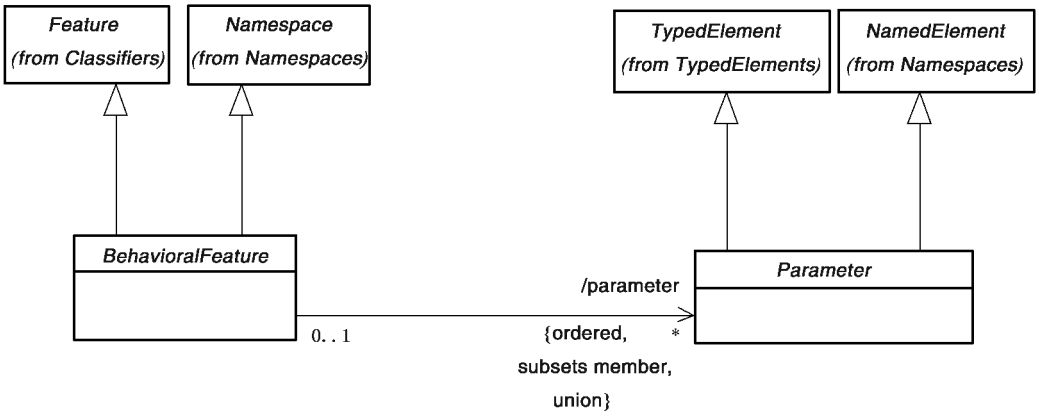


图 14 行为特征包中定义的元素

7.1.1 行为特征(BehavioralFeature)

行为特征是类目的一个特征,其定义了实例的行为特征。

描述

行为特征是规定其实例的行为方面的类目的一个特征。行为特征(BehavioralFeature)是将特征 Feature 和命名空间 Namespace 特化的一个抽象元类。行为方面的种类由 BehavioralFeature 的子类进行建模。

属性

无附加属性。

关联

/parameter:Parameter[*]——定义行为特征类 BehavioralFeature 的参数,它是 Namespace::member 的子集,是一个有序的派生联合。

约束

无附加约束。

附加操作

[1] 查询 isDistinguishable()确定两个 BehavioralFeatures 类是否可在同一命名空间共存。它规定两者的特征标记应不同。

```
BehavioralFeature::isDistinguishableFrom(n:NamedElement,ns:Namespace):Boolean;
isDistinguishableFrom=
if n.oclIsKindOf(BehavioralFeature)
then
    if ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->notEmpty()
    then Set{}->including(self)->including(n)->isUnique(bf|bf.parameter->collect(type))
    else true
    endif
else true
endif
```

语义

参数列表描述了可在启用 BehavioralFeatures 时给定的变元的顺序和类型。

记法

无附加记法。

7.1.2 参数(Parameter)

参数是一种为行为特征的启用传入传出信息的变元的规约。

描述

参数 **Parameter** 是将类型化元 TypedElement 和命名元 NameElement 特化的抽象元类。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

参数规定为同操作一样的行为元素的启用传入传出的变元。参数的类型限定所能传递的值。

一个参数可赋予一个名称,名称以此在一个行为特征各参数中惟一标识该参数。当参数未被命名时,只能由其在参数的有序列表中的位置区分。

记法

无通用的记法。行为特征类 BehavioralFeatures 的特定子类会为它们的参数定义记法。

样式指南

参数名通常以小写字母开头。

7.2 可更改性包(Changeabilities)

抽象包的可更改性子包 changeabilities 定义某个结构特征何时可由客户进行修改。可更改性包如图 15 所示,可更改性包中定义的元素如图 16 所示。

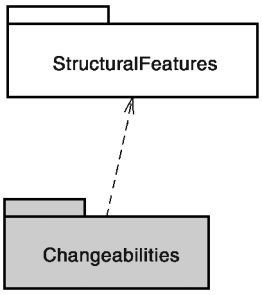


图 15　可更改性包(changeabilities)

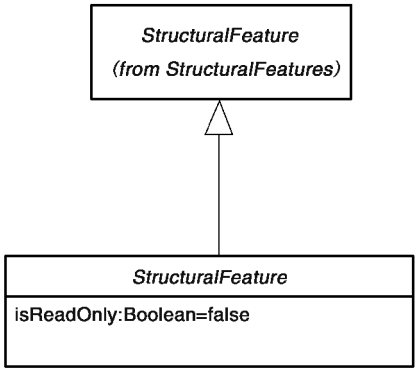


图 16　可更改性子包(changeabilities)中定义的元素

7.2.1　结构特征(特化的)　[StructuralFeature (specialized)]

描述

通过对结构特征 StructuralFeature 的特化,添加了一个确定客户可否修改其值的属性。

属性

isReadOnly:Boolean——说明特征的值可否由客户修改。默认为 false。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

只读的结构特征由{readOnly}表示为结构特征记法的一部分。可修改的结构特征由{unrestricted}表示为结构特征记法的一部分。这种标注可加以抑制,这时从图中不能确定其值。

表示选项

只有当 isReadonly=false 时,才可以抑制这种注记。这时,可以认为所有没有标注{readOnly}的情形其值都是 false。

7.3　类目包(Classifiers)

抽象包中的类目包 Classifiers 为了能够把实例根据不同的特征分类,通过泛化关系定义了两个抽象类。如图 17、图 18 所示。

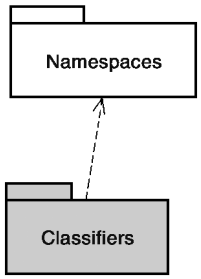


图 17　类目包

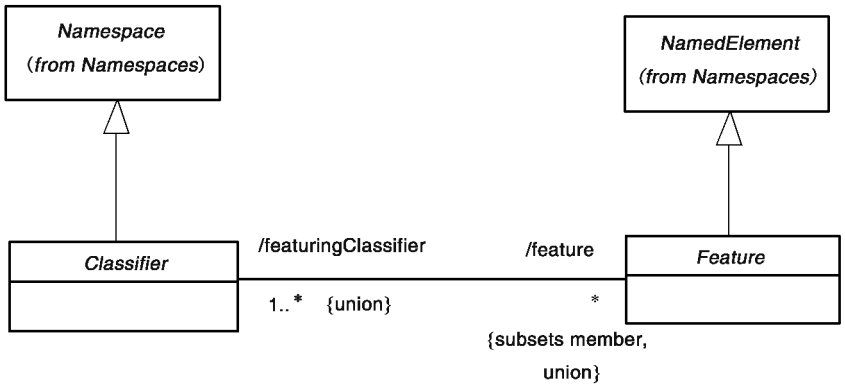


图 18　类目包中定义的元素

7.3.1　类目(Classifiers)

类目是实例的一种分类——它描述了一组具有共同特征的实例。

描述

类目是一种其成员可包含特征的命名空间。类目 Classifiers 是一个抽象元类。

属性

无附加属性。

关联

/feature:Feature[*]——规定在类目中定义的每一个特征。是 Name::member 的子集,一个派生联合。

附加操作

[1]　查询 allFeatures()给出类目命名空间内的所有特征。一般来讲,通过继承之类的机制,该操作的结果集比原特征要大。

Classifier::allFeatures():Set(Feature);

AllFeatures=member->select(ocllsKindOf(Feature))

约束

无附加约束。

语义

一个类目是按特征对实例的一种分类。

记法

类目的默认记法是用实边线的矩形,其中包含类目的名称。另外一种可选的记法是,该矩形已用横线隔成小格,内含特征或该类目的其他成员。类目具体类型可以用双尖号标记在名称的上方。有些类目的特化有自己独特的记法。

表示选项

任何格子都可加以抑制。被抑制的格子不画分隔线。当某个格子被抑制时,无法推断其中出现哪些元素。必要时,可用格子名称来消除这种歧义性。

7.3.2 特征(Feature)

一个特征声明类目实例的行为或结构特性。

描述

一个特征声明类目实例的行为或结构特性。特征是一个抽象元类。

属性

无附加属性。

关联

/featuringClassifier[1..*]——以此 Feature 作为一个特征类目。是一个派生联合。

约束

无附加约束。

语义

特征表示了它所描述的类目的一些性质。一个特征可以是多个类目的特征。

记法

无通用的记法。子类定义其特定记法。

7.4 注释包(Comments)

抽象包中的注释包定义了为任意元素附加注释的通用能力。如图 19、图 20 所示。

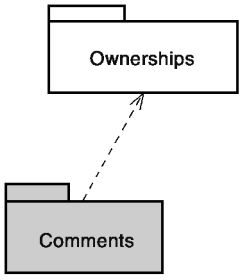


图 19 注释包

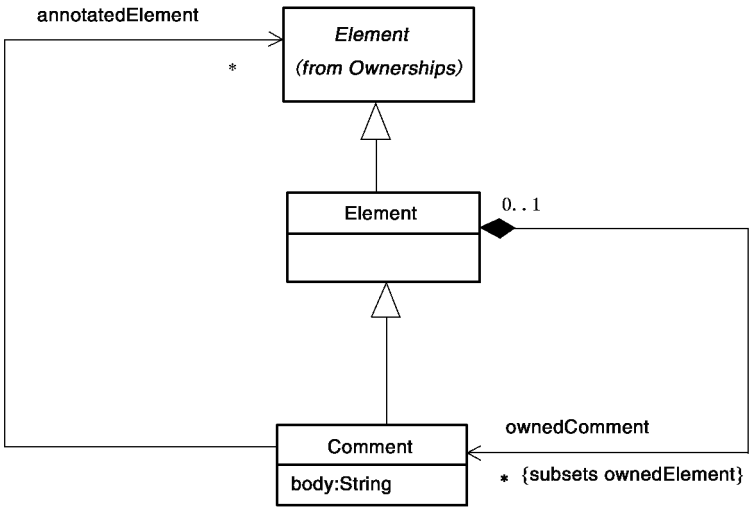


图 20 注释包中定义的元素

7.4.1 注释(Comment)

注释是能够附加到一组元素上的文本标注。如图 21 所示。

描述

注释使得能够向元素添加各种各样的评注。注释没有语义作用,但是可包含对建模者有用的信息。

一个注释可以由任何元素使用。

属性

Body:String——规定作为该注释的串。

关联

annotatedElement:Element[*]——引用被注释的 Element(s)。

约束

无附加约束。

语义

注释对所标注的元素不增加任何语义,但可表示对模型的读者有用的信息。

记法

注释由一个右上角折叠的长方形(称为"注符")所示。长方形包含注释体。注释和每一所注释的元素用虚线连接。

表示选项

在语境明确或在图中无关紧要时,连接注释和所注释元素的虚线可以略去。

示例

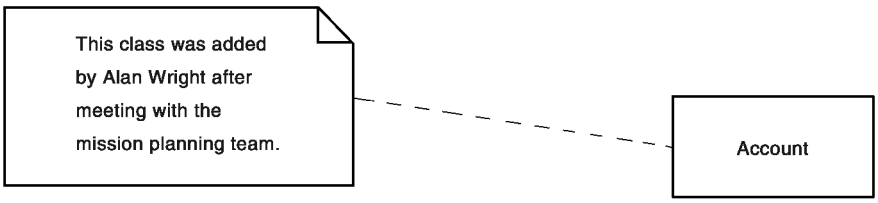


图 21 注释标注

7.4.2 元素 [Element(特化的)]

描述

一个元素可有多个注释。

属性

无附加属性。

关联

ownedComment:Comment[*]——该元素所拥有的注释。是 Element::ownedElment 的子集。

约束

无附加约束。

语义

元素的注释不增加任何语义,但是可表示对模型的读者有用的信息。

记法

无附加记法。

7.5 约束包(Constraints)

抽象包的约束 Constraints 子包规定可以用来向元素增加语义信息的基本构建块。如图 22 和图 23 所示。

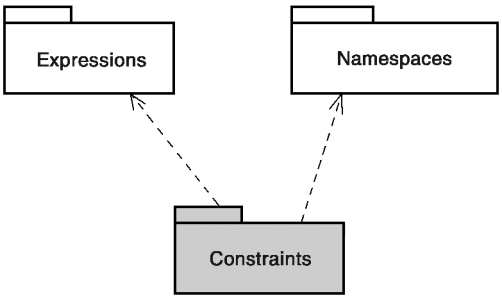


图 22 约束包

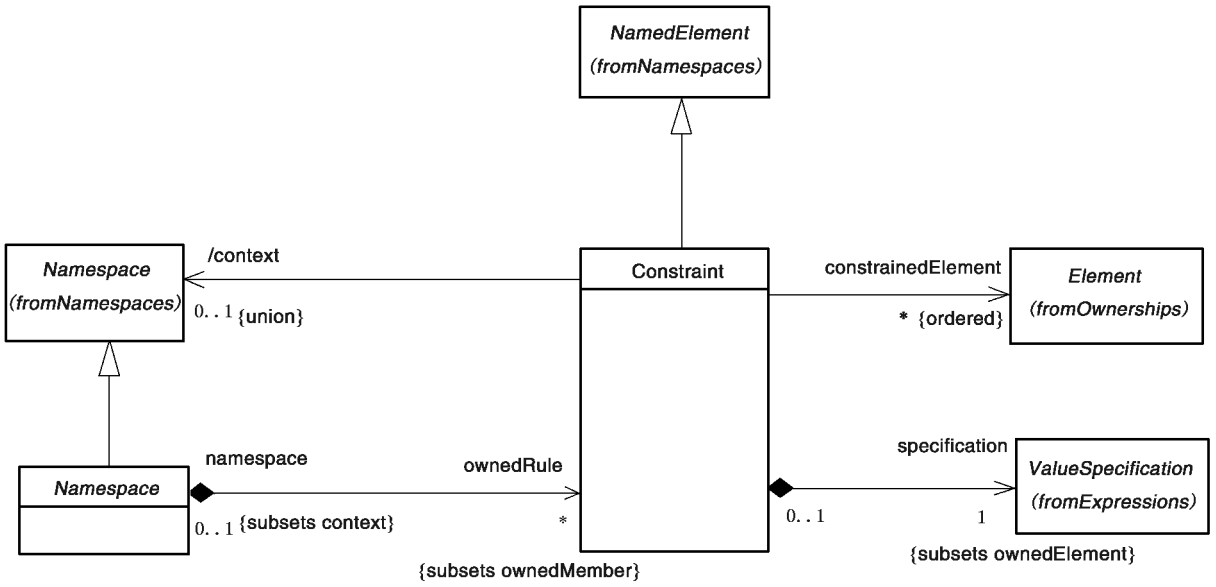


图 23 约束包中的定义的元素

7.5.1 约束(Constraints)

约束是一种条件或限定,它以自然语言的文本或机器可读的语言表达,目的是声明某个元素的一些语义。

描述

约束包含一个用来定义一个或多个元素的附加语义值规约 ValueSpecification。UML 预先定义了几种约束(比如关联"xor"约束),其他的可由用户定义。用户定义的约束以规定的语言描述,其语法和解释由工具完成。一种编写约束的预定义的语言就是 OCL。有些情况下,编程语言(如 Java)适合于表达约束。在其他情况下,可采用自然语言。

约束是一种条件(一个布尔表达式),用于限制其他语言构造对关联元素的外延。

约束有一个可选名称,但通常不用命名名称。

属性

无附加属性。

关联

constrainedElement：Element[*]——此约束所引用的一组有序的元素。

/context：Namespace[0..1]——规定作为对此约束求值的语境的命名空间。这是一个派生的联合。

Specification：ValueSpecification[0..1]——为使该约束得到满足而求值应为 true 的条件。是 Element::ownedElement 的子集。

约束

[1] 约束的取值规约应求出一个布尔值。

不能以 OCL 表达。

[2] 对约束的取值规约求值时一定不能带来副作用。

不能以 OCL 表达。

[3] 约束不能施加于本身。

tot constrainedElement->includes(self)

语义

约束 Constraints 表示连接在受约束元素上的附加语义信息。一个约束就是一个断言，指明对系统的正确设计所应满足的限制。受约束元素是对约束规约求值所需的元素。另外，对约束 Constraints 的语境进行访问，并可作为对该规约对名称进行解释时的命名空间。例如 OCL 中“self”是对语境元素的引用。

在某些语言里，约束常以文本串表达。当采用类 OCL 等形式语言时，可以用工具来验证约束的某些方面。

一般来讲，一个约束可能属于多种元素。惟一的限制是属主元素应该对被约束元素有访问权。

约束 Constraints 的属主将确定何时对约束表达式求值。例如，可以由一个操作 Operation 来规定一个约束 Constraints 表示的是前置条件还是后置条件。

记法

约束按照下面的 BNF 形式由花括号内的文本串给出。

Constraint::='{'[<name>':']<Boolean expression>'}'

对于其记法是文本串(如一个属性)的元素，约束串跟在花括号中的元素文本串之后。图 24 所示为约束紧跟在类别符号内的属性的情形。

对于施加于单一元素(如一个类或者一个关联路径)的约束，约束串可位于元素字符的近旁，并最好靠近元素的名称(如果有时)。建模工具应能确定被约束的元素。

对于应用于两个元素(如两个类或两个关联)的约束，约束由两个元素中间标有约束文本串(在花括号内)的虚线表示。图 25 所示为了两个关联之间的{xor}约束。

表示选项

约束串可以放在标注符号中，并以虚线连接到每一个被约束的元素的符号。图 26 所示是采用标注符号的约束的例子。

当约束以两元素之间的虚线所示时，箭头可置于其中一端。箭头指向约束中的有关信息。在受约束元素集合中，处在箭头尾部的元素放在第一个位置上，处在箭头头部的元素放在第二个位置上。

对于相同类型的三条或更多条路径(比如泛化路径或关联路径)，约束可连接到跨所有路径的虚线。

示例

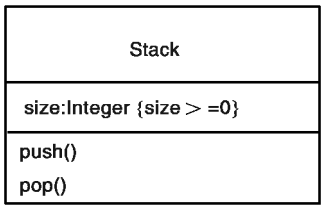


图 24　附于属性的约束

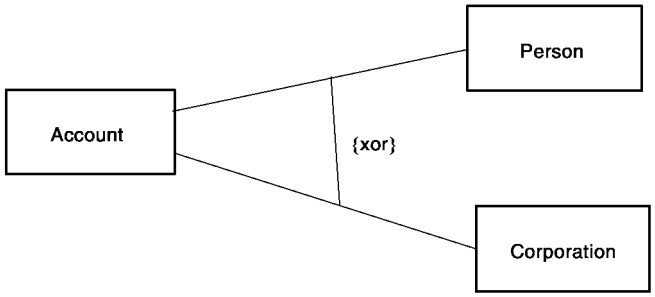


图 25　{xor}约束

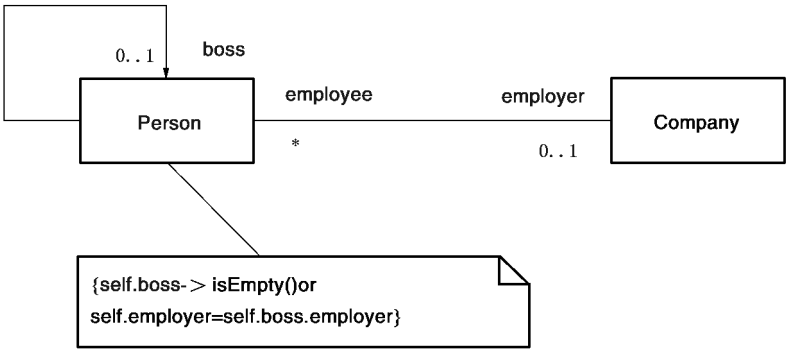


图 26　加注释符号中的约束

7.5.2　命名空间[Namespace(特化的)]

描述

命名空间可以拥有约束。约束不一定施加于命名空间本身,还可施加于命名空间中的元素上。

属性

无附加属性。

关联

ownerRule:Constrain[*]——规定属于命名空间的约束集。是 Namespace::ownedMember 的子集。

约束

无附加约束。

语义

命名空间 Namespace 的约束 ownedRule 表示被约束元素的良构性规则。当确定模型元素是否是良构时,对这些约束求值。

记法

无附加记法。

7.6 元素包(Elements)

抽象包的 **Elements** 子包定义多数基本的抽象类和元素。如图 27、图 28 所示。

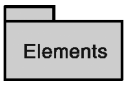


图 27 元素包

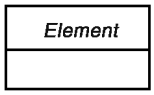


图 28 元素包中定义的元素

元素

元素是模型的组成部分。

描述

元素是没有超类的抽象元类。在基础结构库中,它用作所有元类的公共超类。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

元素类的子类为所表示的概念提供适当的语义。

记法

元素类没有通用的记法。元素类的特定子类定义各自的记法。

7.7 表达式包(Expressions)

抽象包中的表达式包 Expressions 规定支持取值规约的通用的元类,并通过对 Expressions 的特化,还进一步支持结构化的表达式树及不透明的或非解释性表达式。各种 UML 的结构中都需要或使用表达式,表达式是在语境中取值时产生值。如图 29、图 30 所示。

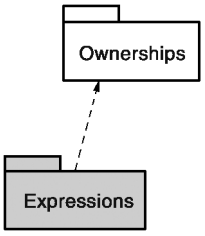


图 29 表达式包

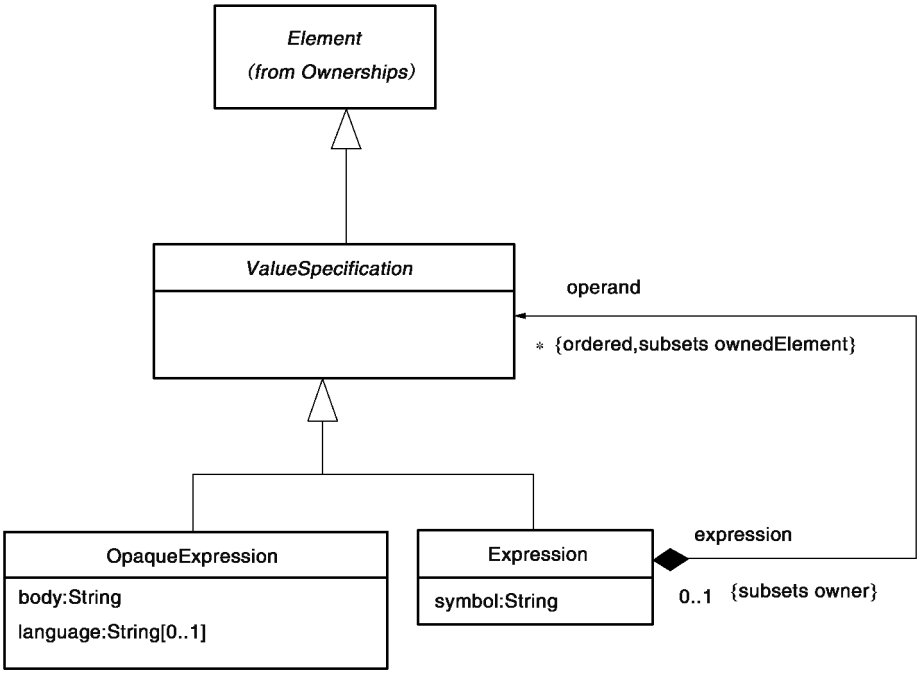


图 30 表达式包中定义的元素

7.7.1 表达式

表达式是符号的结构化树,在语境中求值时指代值集(可能为空)。

描述

表达式表示一个表达式树中的一个节点,可以是非终结节点或者是终结节点。表达式定义了符号,而可能作取值规约的操作数的空序列。

属性

symbol:String[1]——与表达式数中和节点相关联的符号。

关联

operand:ValueSpecification[*]——规定操作数的序列。是 Element::ownedElement 的子集。

约束

无附加约束。

语义

表达式表示表达式树中的一个节点。当没有操作数时,代表一个终结节点。当有操作数时,就表示施加到那些操作数的操作符。不论哪种情况,都有一个符号和节点关联。对这一符号的解释取决于表达式的语境。

记法

在默认情况下,没有操作数的表达式由它的符号标记即可,不需引号。有操作数的表达式由它的符号跟以由圆括号括起的按顺序排列的操作数来标记。在特定的语境中,允许采用专用记法,包括中缀操作符。

示例

xor

else

plus(x,1)

x+1

7.7.2 不透明表达式(OpaqueExpression)

不透明表达式是非解释性的文本语句,在语境中求值时指表一个值集(可能为空)。

描述

不透明表达式包含一个语言特有的文本串,用来描述单个值或者多个值及可选的语言规约。

OCL 是规定表达式的一种预定义语言。也可采用自然语言或编程语言。

属性

body:String[1]——表达式的文本。

language:String[0..1]——规定陈述表达式的语言。对表达式体的解释取决于这个语言。如果对语言未作规定,则它可能隐含于表达式体或语境中。

关联

无附加关联。

约束

无附加约束。

语义

对表达式体的解释取决于语言。如果对语言未作规定,则它可能隐含于表达式体或语境中。

假定所规定语言的语言学分析器对表达式体求值。什么时候求值不作规定。

记法

在特定语言中,不透明表达式表示成文本串,串的语法由工具和语言的语言学分析器负责。

不透明表达式是包含它的元素的记法的一部分。

当对不透明表达式的语言作出规定时,一般不在图中标出。一些建模工具可能利用特定的语言,或取用特定的默认语言。一般来说,通过使表达式形式的目的清楚就隐含所采用的是什么语言。如果要图示出语言的名称,则要放在表达式串之前的花括号({})中。

样式指南

语言名称的拼写和大写要准确无误,和定义语言的文档中的一致。比如使用 OCL 而不使用 ocl。

示例

a> 0

{OCL}I>j and self.size>i

average hours worked per week

7.7.3 值规约(ValueSpecification)

值规约 ValueSpecification 是对一个实例集(可能为空)的规约,其中实例包括对象和数据的值。

描述

值规约 ValueSpecification 是一个抽象元类,用来标识模型中的一个或多个值。它可引用一个实例,或者可能是在求值时指代一个或多个实例的表达式。

属性

无附加属性。

关联

expression:Expression[0..1]——如果这个值规约是个操作数,则表示拥有者表达式。是 Element::owner 的子集。

约束

无附加约束。

附加操作

下面介绍一些附加操作。它们将在子类中重新定义。与标准一致的实现能够为包含这些操作的约束所规定的表达式求值。

[1] 查询操作 isComputable()确定在模型中某个值规约能否加以计算。这一操作不能在 OCL 中完全定义。对可计算的所有的值规约,一致的实现该得出一个 true 值;并对操作为 true 的所有值进行计算。一致的实现应能计算所有文字的值。

ValueSpecifivation::isComputable():Boolean;

IsComputable=false

[2] 查询 integerValue()在可计算时给出单一的整数值。

ValueSpecification::BooleanValue():[integer];

integerValue=Set{}

[3] 查询 Boolean::booleanValue()在可计算时给出一个布尔值。

ValueSpecification::booleanValue():[Boolean];

booleanValue=set{}

[4] 查询 stringValue()在可计算时给出单一的串值。

ValueSpecification::stringValue():[String];

stringValue=Set{}

[5] 查询 unlimitedValue()在可计算时给出一个无穷的自然数值。

ValueSpecification::unlimitedValue():[UnlimitedNatural];

unlimitedValue=Set{}

[6] 查询 isNull 在该值为 null 可计算时返回为真。

ValueSpecification::isNull():Boolean;

isNull=false

语义

值规约输出 0 个、1 个或多个值。值的类型和数目要求要符合值规约所在的语境。

记法

无特定记法。

7.8 泛化包(Generalizations)

抽象包中的泛化包 Generalizations 提供了一种定义类之间泛化关系的机制。如图 31、图 32 所示。

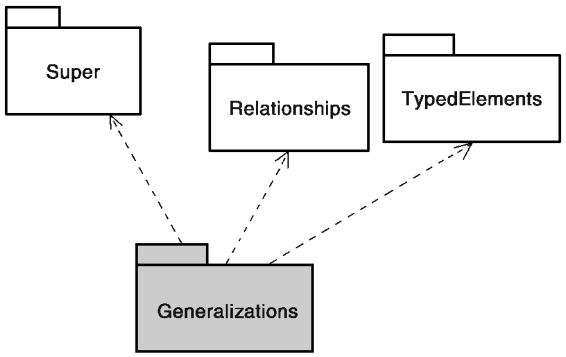


图 31 泛化包

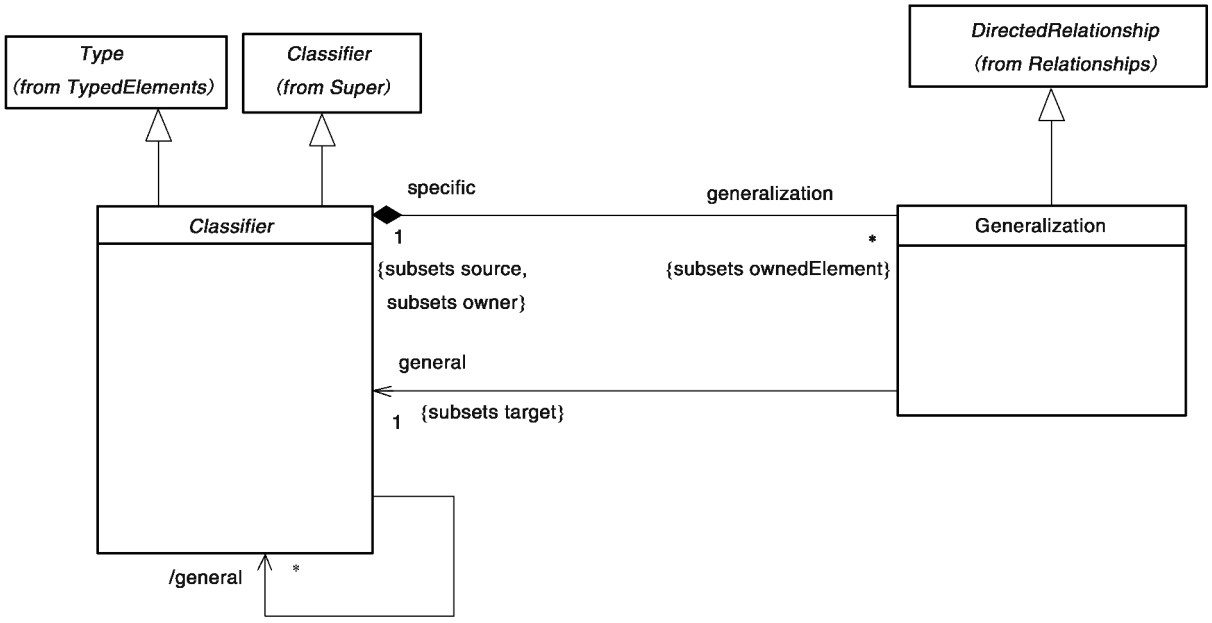


图 32　泛化包中定义的元素

7.8.1　类目[Classifier(特化的)]

描述

一个类目是一种类型,并可拥有泛化;因此能定义对其他类目的泛化关联。

属性

无附加的属性。

关联

/generalization:Generalization[*]——规定该类目 Classifier 的泛化关联 Generalization。在泛化层次中,这些泛化类导航到更为通用的类目。是 Element::ownedElement 的子集。

/general:Classifier[*]——规定该类目的通用类目。这是派生的。

约束

[1]　通用类目是通过泛化关联引用的类目。

general=self.parents()

附加操作

[1]　查询 parents()给出一个泛化的类目的所有直接祖先。

Classifier::parents():Set(Classifier);

parents=generalization.general

[2]　查询 ConformsTo()为定义符合另一类目的类型的类目时给出 true 值。例如在规定特征标记对操作的一致性时就用到该查询。

Classifier::conformsTo(other:Classifier):Boolean;

conformsTo=(self=other)or(self.allParents()->includes(other))

语义

一个类目可以参与对其他类目的泛化关联。一个特定类目的实例也是一个通用类目的(间接的)实例。关于泛化如何影响类目的具体子类型的特定语义是变化的。

一个类目定义一种类型。可泛化类目之间定义了类型的一致性,以使一个类目与自身而且与该泛化层中的所有祖先都是一致的。

记法

无附加的记法。

示例

见泛化 Generalization 类。

7.8.2 泛化(Generalization)

泛化类是在一个较通用类目与一个较特定类目之间的分类学关系。每个特定类目的实例也都是通用类目的实例。因此,特定类目间接地具有通用类目的特征。

描述

泛化建立起一个通用类目和一个特定类目的关系,并属于特定类目。

属性

无附加的属性。

关联

general:Classifier[1]——引用泛化关联中的通用类目。是 DirectedRelationship::target 子集。

specific:Classifier[1]——引用泛化关联中的特定类目。是 DirectedRelationship::source 和 Element::owner 的子集。

约束

无附加的约束。

语义

凡是泛化建立起特定类目与通用类目关系之处,每一个特定类目的实例也都是那个通用类目的实例。因此,为每个通用类目的实例所规定的性质也都隐式对特定类目的实例作了规定。所有施加在通用类目实例上的约束也都施加在特定类目上。

记法

一个泛化类图示为一个带中空的三角形箭头的线段,连接表示所涉及类目的符号。箭头指向表示通用类目的符号。这种记法称为"独自目标样式"(见后面的例子)。

表示选项

引用同一个通用类目的多个泛化关联可以用"共享目标样式"连到一起。见后面的例子,如图 33 所示。

示例

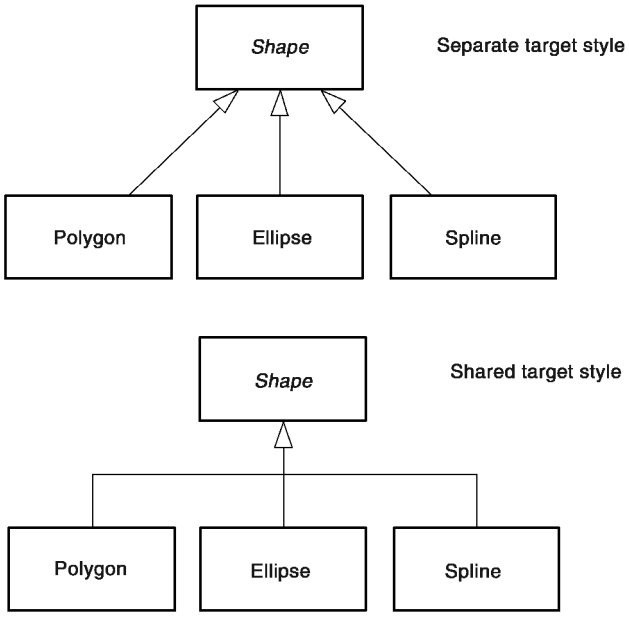


图 33 类之间的泛化例子

7.9 实例包(Instances)

抽象包中实例包 Instances 提供类目的建模实例,如图 34、图 35 所示。

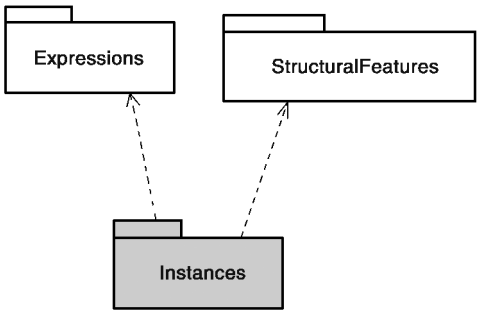


图 34 实例包

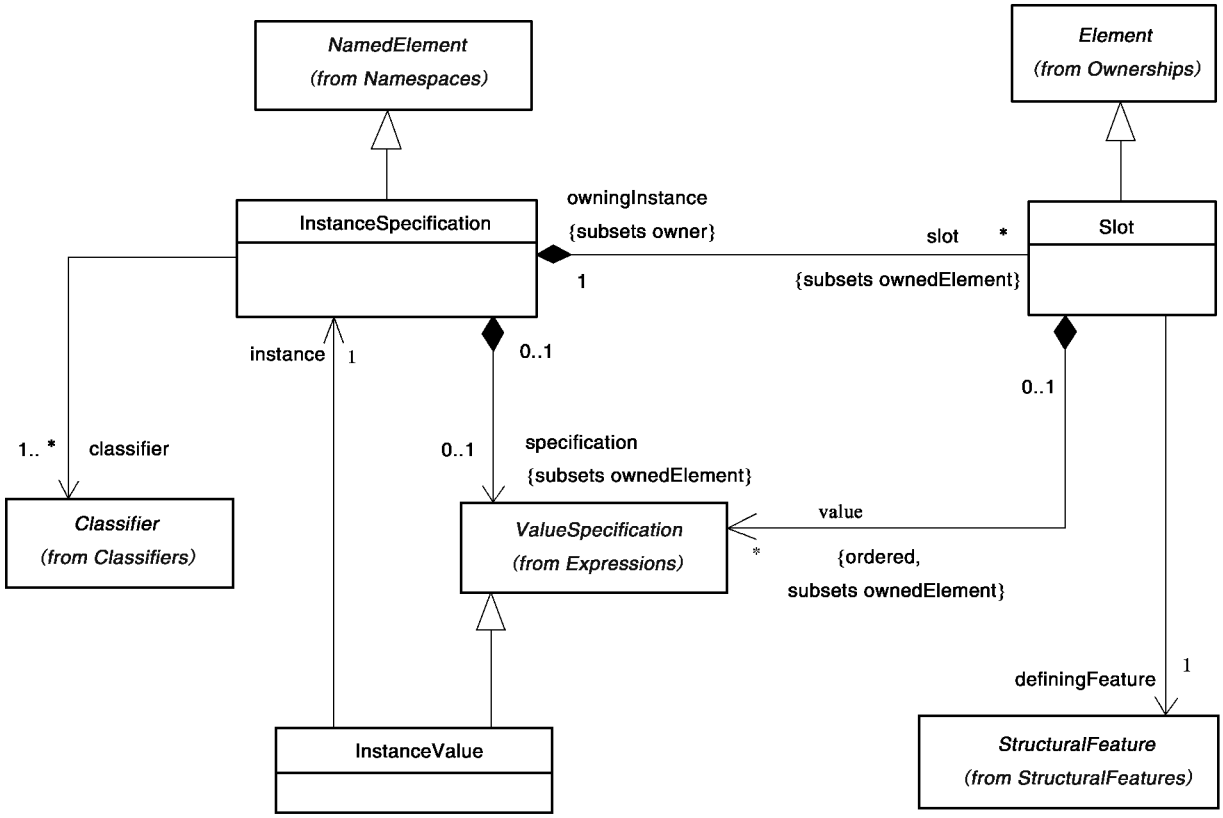


图 35 实例包中定义的元素

7.9.1 实例规约(InstanceSpecification)

实例规约是一种表示被建模的系统中的实例的模型元素。

描述

一个实例规约规定被建模系统中一个实体的存在,并且完全地或部分地描述该实体。描述的内容包括:

a) 对实体的分类:按照实体是一实例的一个或多个类目进行。如果仅有的规定类目是抽象的,那么实例规约只对该实体部分加以描述。

b) 基于其一个或多个类目的实例的种类——例如，其类目是一个类的实例规约描述该类的一个对象，而其类目是一个关联的实例规约描述该关联的一个链接。

c) 实体结构特征的取值规约。并不是实例规约的所有类目的所有结构特征都需要用槽来描述，这时实例规约只是一种部分描述。

d) 关于怎样计算、派生或构造实例的规约(可选)。

实例规约 InstanceSpecification 是一个具体的类。

属性

无附加的属性。

关联

a) classifier:Classifier[1..*]——所表示实例的一个或多个类目。如果规定多个类目，则该实例是按全部类目来分类。

b) slot:Slot[*]——一个槽给出了实例的一个结构特征的(一个或多个)值。一个实例规约为其类目的每一个结构特征(包括继承的特征)都建立一个槽。不必为每一个结构特征建立一个槽，此时，实例规约是一种部分描述。是 Element::ownedElemnt 的子集。

c) specification:ValueSpecification[0..1]——关于怎样计算、派生或构造实例的规约。

约束

[1] 每个槽定义的特征都是实例规约中类目的一个结构特征(直接的或继承的)。

slot->forAll(s|classifier->exists(c|c.allFeatures()->includes(s.definingFeature)))

[2] 一个结构特征(包括从多个类目中继承的同一特征)最多是实例规约中的一个槽的定义特征。

classifier->forAll(d|slot->selects(s|s.definingFeature=f)->size()<=1)

语义

一个实例规约可规定被建模系统中一个实体的存在。实例规约可以提供被建模系统中的可能实体的一个说明或例子。实例规约描述实体，但描述的细节可能不完全。实例的目的是指出实体在被建模系统中所感兴趣的内容。实体与实例规约中每一个类目的规约都一致，并据有带由实例规约的各槽指明的值的特征。某些特征在实例规约中没有槽并不意味着所表示实体没有这个特征，而仅仅是说明该特征在模型中不感兴趣。

一个实例规约能够表示实体在某一瞬时状态(即快照)。对实体的改变可以用多个实例规约来建模，每次改变都有一个快照。

注：在一个被建模系统中，当 n 用来提供一个实体的例子或说明时，实例规约类 InstanceSpecification 并不描述精确的运行时结构，而只是描述了这些结构的信息。对运行时结构的实现细节，不能从中得出任何结论。当用来规定被建模系统中一个实体的存在时，实例规约表示该系统的一部分。对实例规约可建立不完全的模型——即使一个实际的实体有具体的分类，所需的结构特征也可以省略，且实例规约的类目也可以是抽象的。

记法

实例规约记法与其类目采用相同记法，但在原来类目名的地方换成实例的名称(如果有的话)跟以一个冒号(":")和一个或多个类目名。这个名称串带有下划线。如果有多个类目，则全部写出，并用逗号将各名称隔开。类目名在图中可以省略。

如果一个实例规约有一个值规约，值规约写在紧跟名称的等号之后，或者不用等号而将其放在名称的下边。如果实例规约用一个带名称的封闭图形(如矩形)表示，则将值规约写在这个封闭图形之中。如图 36 所示。

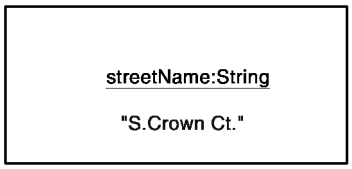


图 36　串实例的规约

槽用类似于对应结构特征的记法图示。在特征可以在格子中以文本的形式图示时，该特征的槽图示成如下文本串：特征的名称跟一个等号和一个值规约。特征的其他性质(例如它的类型)的图示可选。如图 37 所示。

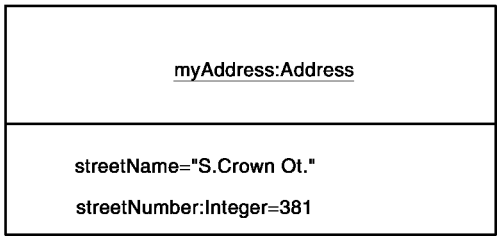


图 37　带有值的槽

其类目是一个关联的实例规约表示一个链接，并和关联一样的记法图示，但实线连接的是实例规约而不是类目。当从与实例规约的连接上就可清楚地看出它表示一个连接而不是一个关联时，就不必再写上带下划线的名称。关联端可以写上名称。导航箭头可以标上，但标上时应符合关联端的导航方向。如图 38 所示。

图 38　表示由一个链接相连的两个对象的实例规约

表示选项

一个属性的槽值可采用类似链接的记法图示。从拥有实例规约到目标实例规约的表示槽值的实线和属性名称标在路径的目标端。如果表上有可导航性时，导航方向一定指向目标。

7.9.2　实例值(InstanceValue)

实例值是标识实例的值规约。

描述

实例值规定由一个实例规约建模的值。

属性

无附加的属性。

关联

instance：InstanceSpecification[1]——作为规定值的实例。

约束

无附加的约束。

语义

实例规约是规定的值。

记法

实例值可以用文本或者图形记法图示。当用文本图示时，可以按属性槽值的方式，并给出实例的名称。当采用图形形式时，以通过连接到实例来表示一个引用值。见“实例规约”。

7.9.3 槽(Slot)

槽规定由实例规约建模的实体具有特定结构特征的一个或多个值。

描述

槽由一个实例规约拥有。它规定定义特征的一个或多个值，即一定是拥有这个槽的实例规约的类目的一个结构特征。

属性

无附加的属性。

关联

a) definingFeature：StructuralFeature[1]——规定可由槽保有的值的结构特征。

b) owningInstance：InstanceSpecification[1]——拥有这个槽的实例规约。Element. owner 的子集。

c) value：InstanceSpecification[*]——对应于定义拥有实例的特征的一个或多个值。这是一个有序的关联。Element. ownerElement 的子集。

约束

无附加的约束。

语义

槽与实例规约、结构性质和一个或多个值有关。它表示由实例规约建模实体具有带规定的一个或多个值的结构特征。在槽中的值应与定义的槽的特征相一致(在类型上，势上等)。

记法

见“实例规约”。

7.10 文字包(Literals)

抽象包中的文字包 Literals 规定用于规定文字值定义的元类，如图 39、图 40 所示。

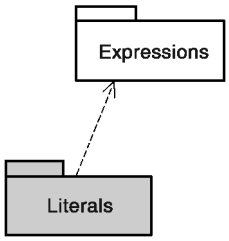


图 39 文字包

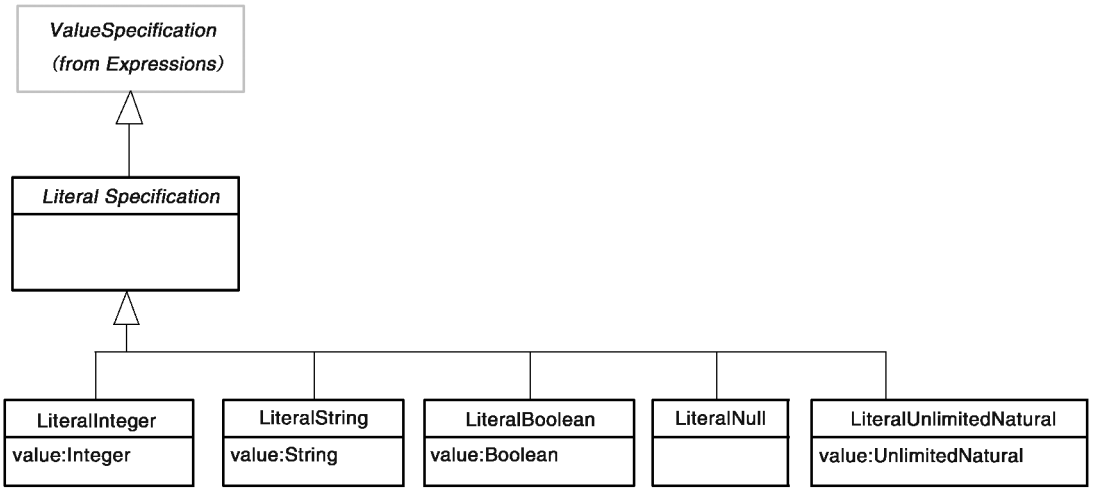


图 40　元素在字符包中的实现

7.10.1　布尔文字类(LiteralBoolean)

布尔文字是布尔值的规约。

描述

布尔文字包括一个布尔值的属性。

属性

value:Boolean——规定的 Boolean 值。

关联

无附加的关联。

约束

无附加的约束。

附加操作

[1]　查询 isComputable()重定义为真。

LiteralBoolean::isComputable():Boolean;

isComputable=true

[2]　查询 booleanValue()给出值。

LiteralBoolean::booleanValue():[Boolean];

booleanValue=value

语义

布尔文字 LiteralBoolean 规定一个布尔常量值。

记法

布尔文字根据其值用以“true”或“false”示出。

7.10.2　整数文字(LiteralInteger)

整数文字是对整数值的规约。

描述

整数文字类包含一个值为整数的属性。

属性

value:Integer——规定的 Integer 的值。

关联

无附加的关联。

约束

无附加的约束。

附加操作

[1] 查询 isComputable()重定义为真。

LiteralInteger::isComputable():Boolean;

isComputable=true

[2] 查询 integerValue()给出值。

LiteralInteger::integerValue():[Integer];

integerValue=value

语义

LiteralInteger 规定常数整数值。

记法

整数文字 LiteralInteger 通常图示成一个数字序列。

7.10.3 空值文字(LiteralNull)

空值文字规定为缺值。

描述

空值文字用来表示空值,即没有值。

属性

无附加属性。

关联

无附加的关联。

约束

无附加的约束。

附加操作

[1] 查询 isComputable()重新定义为 true。

LiteralNull::isComputable():Boolean;

isComputable=true

[2] 查询 isNull()返回 true。

LiteralNull::isNull():Boolean;

isNull=true

语义

空值文字类 LiteralNull 用来为缺值的情况建模。

记法

空值文字类 LiteralNull 的记法根据使用场所不同而变化。通常表示成单词“null”。但也用其他记法表示特定用法。

7.10.4 文字规约(LiteralSpecification)

文字规约标识别在建模中的文字类型常量。

描述

文字规约是识别在建模中的文字类型常量的值规约的一个抽象泛化。

属性

无附加的属性。

关联

无附加的关联。

约束

无附加的约束。

语义

无附加的语义。文字规约类 LiteralSpecification 的子类加以定义，以便规定不同类型的文字值。

记法

无特定的记法。

7.10.5 文字串(LiteralString)

文字串 LiteralString 是串值的规约。

描述

文字串包括一个值为串的属性。

属性

value:String——规定的 String 的值。

关联

无附加的关联。

约束

无附加的约束。

附加操作

[1] 查询 isComputable()重新定义为 true。

LiteralString::isComputable():Boolean;

isComputable=true

[2] 查询 stringValue()给出值。

LiteralString::stringValue():[String];

stringValue=value

语义

文字串 LiteralString 规定一个串值常量。

记法

文字串 LiteralString 图示成以双引号括起的字符序列。

所用的字符集不作规定。

7.10.6 不受限文字自然数(Literal Unlimited Natural)

不受限文字自然数是对不受限的自然数的一个规约。

描述

不受限文字自然数包括一个值为不受限自然数的属性。

属性

value:UnlimitedNatural——规定的不受限自然数的值。

关联

无附加的关联。

约束

无附加的约束。

附加操作

[1] 查询 isComputable()重新定义为 true。

LiteralUnlimitedNatural::isComputable():Boolean;

isComputable=true

[2] 查询 unlimitedValue()给出值。

LiteralUnlimitedNatural::unlimitedValue():[UnlimitedNatural];

unlimitedValue=value

语义

不受限文字自然数规定一个不受限的自然数常量。

记法

不受限文字自然数图示成一个数字序列或一个星号(*),其中星号指代没有限制(不是无穷大)。

7.11 势域包(Multiplicities)

抽象包的势域子包 Multiplicities 定义了元模型类,用来支持类型元素(例如关联端和属性)的势域规约,并规定多值元素是否是有序的和惟一的。如图 41、图 42 所示。

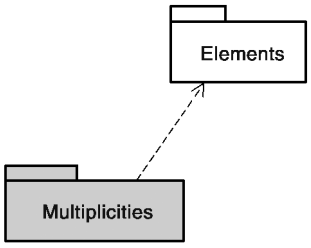


图 41 势域包

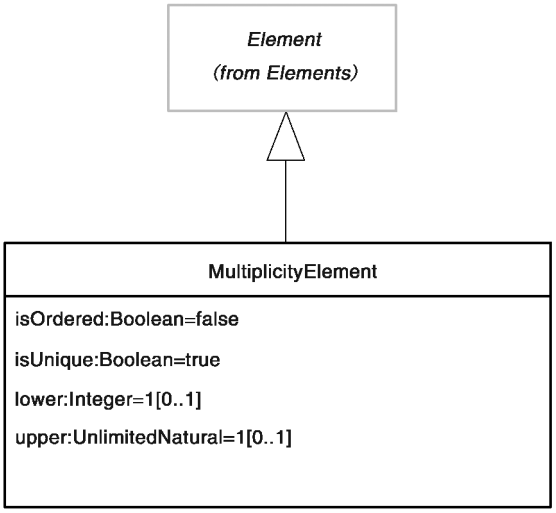


图 42 在势域包里定义的元素

7.11.1 势域元素(MultiplicityElement)

势域是用非负整数定义的闭区间,从下界开始,到上界(可能是无穷大)结束。势域元素嵌入这一信息,以便规定这一元素的例示所允许的势。

描述

势域元素 MuplicityElement 是一个抽象的元类,它包括用来定义势域界可选属性。MuplicityElement 还包括该元素的实例的值是否是惟一的或有序的规约。

属性

a) isOrdered:Boolean——对多值势域,这一属性规定该元素的实例值是否为有序的。缺省为 false。

b) isUnique:Boolean——对多值势域,这一属性规定该元素的实例值是否是惟一的。缺省为 true。

c) lower:Integer[0..1]——规定势域区间的下界。缺省为 1。

d) upper:UnlimitedNatural[0..1]——规定势域区间的上界。缺省为 1。

关联

无附加的关联。

约束

当上界可由模型中不可计算的表达式规定时,这些约束应该对此进行处理。这种情况在这一包中不会出现,但在子类中会出现。

[1] 势域应至少定义一个大于 0 的有效的势。

upperBound()->notEmpty()implies upperBound()>0

[2] 下界应是一个非负整型文字。

lowerBound()->notEmpty()**implies** lowerBound()>=0

[3] 上界应大于或等于下界。

(upperBound()->notEmpty()and lowerBound()->notEmpty())**implies** upperBound()>=lowerBound()

附加操作

[1] 查询 isMultivalued()检验这一势域的上界是否大于 1。

MultiplicityElement::isMultivalued():Boolean;

pre:upperBound()->notEmpty()

isMultivalued=(upperBound()>1)

[2] 查询 includesCardinality()检验规定的势对此势域是否有效。

MultiplicityElement::includesCardinality(C:Integer):Boolean;

pre:upperBound()->notEmpty()and lowerBound()->notEmpty()

includesCardinality=(lowerBound()<=C)and(upperBound()>=C)

[3] 查询 includesMultiplicity()检验这一势域是否包含规定的势域所允许的所有的势。

MultiplicityElement::includesMultiplicity(M:MultiplicityElement):Boolean;

pre:self.upperBound()->notEmpty()**and** self.lowerBound()->notEmpty()

and M.upperBound()->notEmpty()**and** M.lowerBound()->notEmpty()

includesMultiplicity=(self.lowerBound()<=M.lowerBound())**and** (self.upperBound()>=M.upperBound())

[4] 查询 lowerBound()返回一个整型的势域下界。

MultiplicityElement::lowerBound():[Integer];

lowerBound=**if** lower->notEmpty()**then** lower **else** 1 **endif**

[5] 查询 upperBound()返回有界势域的一个不受限自然数的上界。

MultiplicityElement::upperBound():[UnlimitedNatural];

upperBound=**if** upper->notEmpty()**then** upper **else** 1 **endif**

语义

势域定义一个用来对有效的势进行定义的整数。特别地,如果 M. includesCardinality(C)为真,则势 C 对势域 M 是有效的。

势域定义成一个整数区间,从下界开始,到上界(可能是无穷大)结束。

如果一个势域元素 MultiplicityElement 规定为一个多值势域,该元素的例示中有一个值集。势域是对可有效出现于该集合的值数目的约束。

如果 MultiplicityElement 规定为有序的(即 isOrdered 为真),则该元素的例示中的值的集合是有序的。有序隐含着有一个从正整数到该值集元素的映射。如果 MultiplicityElement 不是多值的,则对 isOrdered 的值在语义上没有影响。

如果 MultiplicityElement 规定为无序的(即 isOrdered 为假),则在该元素的例示中的值没有设定顺序。

如果 MultiplicityElement 规定为惟一的(即 isUnique 为真),则该元素的例示中的值集一定是惟一的。如果 MultiplicityElement 不是多值的,则对 isUniqued 的值在语义上没有影响。

记法

MultiplicityElement 的记法由具体的子类定义。一般来说,记法包含区间边界的文本串的势域规约,记法还可选地说明定序和惟一性。

势域边界的一般格式如下:

lower-bound..upper-bound

其中下界 lower-bound 是一个整数,上界 upper-bound 是一个不受限定的自然数。可用字符 * 表示重数的上界为不受限制的自然数(或无穷大)。

当势域与一个用文本串(如属性)表示的元素相关联时,势域串要放在方括号([])内,作为文本串的一部分。图 43 所示为两个势域串表示成一个类符号中属性规约的一部分。

当势域与一个作为符号出现的元素(如关联端点)相关联时,势域串图示不用方括号,可放在该元素符号的附近。图 44 所示为两个势域串图示成两个关联端的规约的一部分。

关于定序和惟一性规约的记法可随势域元素 MultiplicityElement 的特定子类而有所变化。通用记法是采用包含 ordered 或 unordered 的性质串来定义有序性,以包含 unique 或 nonunique 的性质串来定义惟一性。

表示选项

如果下界和上界相等,则采用只包含上界的串这样的替代记法。例如,"1"在语义上等同于"1..1"。

势域的下界为 0 且上界未规定时,可采用包含星号(*)以代替"0..*"这一替代记法。

下面的 BNF 定义了势域串(包括支持表示选项)的句法。

multiplicity::=<multiplicity_range>

multiplicity_range::=[lower'..']upper

lower::=integer

upper::=unlimited_natural|'*'

示例

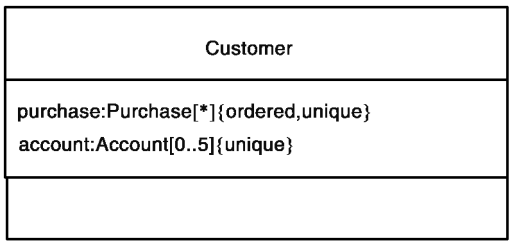


图 43　文本规约中的势域

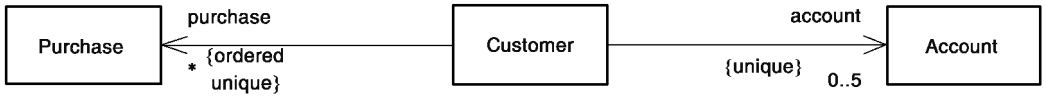


图 44　作为符号标注的势域

论据

势域元素表示一种设计权衡，用来改进某些技术映射（如 XMI）。

7.12　势域表达式包（MultiplicityExpressions）

抽象包中的势域表达式子包 MultiplicityExpressions 扩展了势域的能力，以支持使用上下界的值表达式。如图 45、图 46 所示。

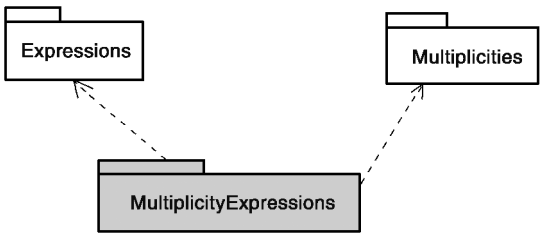


图 45　势域表达式包 MultiplicityExpressions

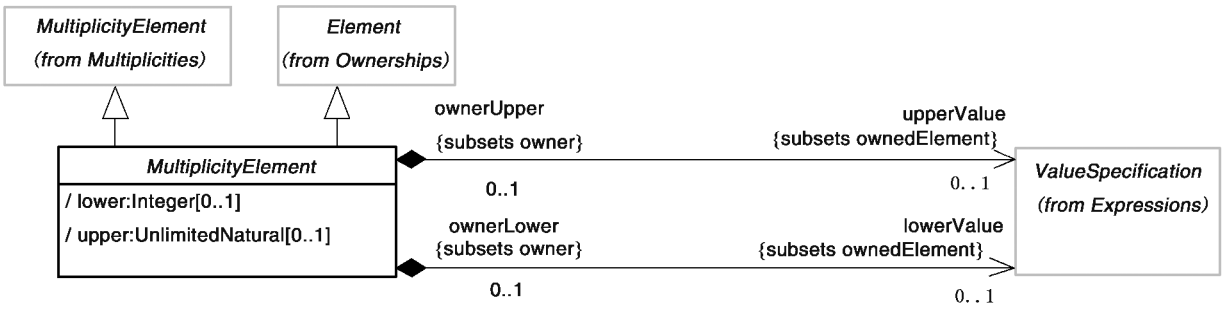


图 46　在势域表达式包中定义的元素

7.12.1　势域元素［MultiplicityElement（特化的）］

描述

势域元素 MultiplicityElement 经过特化，以支持采用值规约来定义势域的上下界。

属性

a）　/lower:Integer[0..1]——当势域区间的下界以一个整数表达时，规定此下界。这是对势域 Multiplicities 相应性质的重定义。

b) /upper:UnlimitedNatural[0..1]——当势域区间相应的性质以一个不受限的自然数表达时，规定此上界。这是对势域 Multiplicities 相应性质的重定义。

关联

a) lowerValue:ValueSpecification[0..1]——势域下界的规约。element::ownedElement 的子集。

b) upperValue:ValueSpecification[0..1]——势域的上界的说明。是 Element::ownedElement 的子集。

约束

[1] 当用值规约 ValueSpecification 作为下界或上界时，对该规约的求值一定不能带来负作用。

不能用 OCL 语言表达。

[2] 当用值规约 ValueSpecification 作为下界或上界时，该规约应是一个常数表达式。

不能用 OCL 语言表达。

[3] 派生的属性 lower 应等于 lowerBound。

lower=lowerBound()

[4] 派生的属性 upper 应等于 upperBound。

upper=upperBound()

附加操作

[1] 查询 lowerBound()返回一个整数型的势域下界。

MultiplicityElement::lowerBound():[Interger];

lowerBound=**if** lowerValue->isEmpty() **then**

1

else

lowerValue.integerValue()

endif

[2] 查询 upperBound()返回一个不受限自然数型的势域上界。

MultiplicityElement::upperBound():[UnlimitedNatural];

upperBound=

if upperValue->isEmpty() **then**

1

else

upperValue.unlilmitedValue()

endif

语义

势域元素 MultiplicityElement 的势域的上下界可由值规约——例如(元副作用的、常量)表达式——来规定。

记法

对记法 Multiplicities::MultiplicityElement 加以扩展，以支持上下界的值规约。

下面的 BNF 定义势域串(包括支持表示选项)的语法。

multiplicity::=<multiplicity_range>['{'<order_designator>'}']

multiplicity_range::=[lower'..']upper

lower::=integer|value_specification

upper::=unlimited_natural|'*'|value_specification

<order_designator>::=ordered|unordered

<uniqueness_designator>::=unique|nonunique

7.13 命名空间包(Namespaces)

抽象包的命名空间子包 Namespaces 规定:用于定义命名名称的模型元素和在命名空间内已命名元素的包和标识的概念。如图 47、图 48 所示。

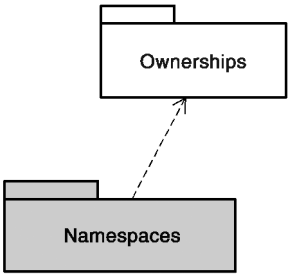


图 47 命名空间包

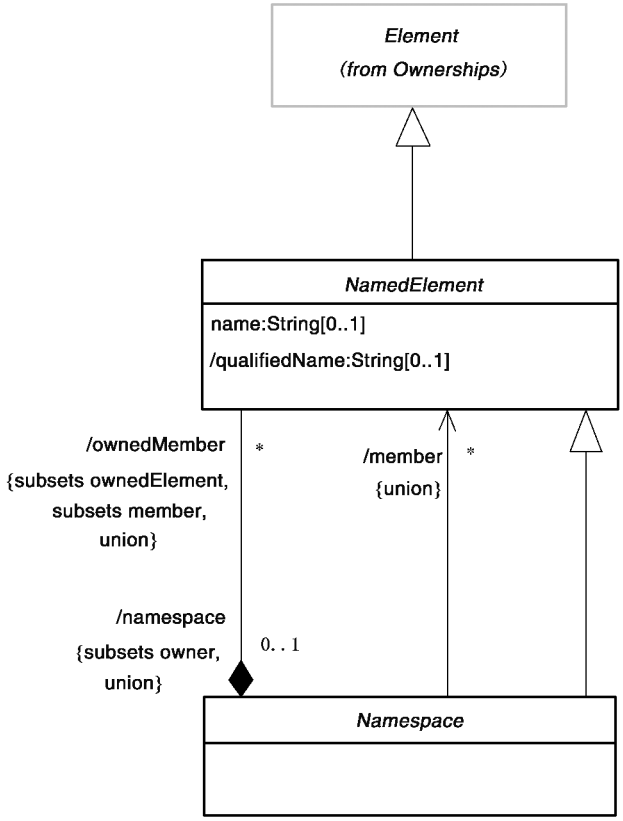


图 48 命名空间包中定义的元素

7.13.1 命名的元素(NamedElement)

命名元素是可命名名称的模型中的元素。

描述

命名元素表示可命名名称的元素。名称用来标识命名元素在其中定义的命名空间中的命名元素。命名元素也有一个限定名。命名元素 NamedElement 是一个抽象元类。

属性

a) name:String[0..1]——命名 NamedElement 的名称。

b) /qualifiedName:String[0..1]——限定名,使得在嵌套的命名空间的层次结构中,能对其标识的名称命名。它由包含命名空间的名称来构造,从该层次的根开始,到命名元素自身结束。这是一个派生属性。

关联

/namespace:Namespace[0..1]——规定拥有该命名元素的命名空间。Element::owner 的子集。这是一个派生联合。

约束

[1] 如果没命名名称或包含的命名空间之一没命名名称时,则没有限定名。

(self.name->isEmpty() **or** self.allNamespaces()->select(ns | ns.name->isEmpty())->notEmpty())

implies self.qualifiedName->isEmpty()

[2] 如果命名名称,且包含该命名的所有命名空间都命名名称时,那么该限定名的用各包含的命名空间的名称来构造。

(self.name->notEmpty() **and** self.allNamespaces()->select(ns | ns.name->isEmpty())->isEmpty())

implies

self.qualifiedName=self.allNamespaces()->iterate(ns:Namespace;result:String=self.name|
ns.name->union(self.separator())->union(result))

附加操作

[1] 查询 allNamespaces()按从内到外的嵌套顺序给出命名元素所在命名空间序列。

NamedElement::allNamespaces():Sequence(Namespace);

allNamespaces=

if self.namespace->isEmpty()

then Sequence{}

else self.namespace.allNamespaces()->prepend(self.namespace)

endif

[2] 查询 isDistinguishableFrom()确定两个命名元素在逻辑上可否共存于一个命名空间之内。在默认情况下,如果两个命名命名元素(a)有不相关的类型,或(b)有相关的类型但名称不同,那么这两个元素是可区分的。

NamedElement::isDistinguishableFrom(n:NamedElement,ns:Namespace):Boolean;

isDistinguishable=

if self.oclIsKindOf(n.oclType)**or** n.oclIsKindOf(self.oclType)

then ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()

else true

endif

[3] 查询 separator()给出在构造限定的名称时用于分隔名称的串。

NamedElement::separator():String;

separator='::'

语义

名称属性用于标识其名称可访问的命名空间中的命名元素。注意该属性有一个[0..1]的势域,提供不给出名称的可能性(这不同于空名称)。

7.13.2 命名空间(Namespace)

命名空间是包含一组可通过名称标识的命名元素的模型中的元素。

描述

命名空间是一个可以拥有其他命名元素的一个命名元素。每一个命名元素至多可属于一个命名空间。

命名空间提供一种通过名称来标识命名元素的方法，命名元素可以通过命名空间中的名称进行标识，或者直接属于命名空间，或者由其他方法(例如引入或继承)来引进。命名空间是一个抽象元类。

属性

无附加的属性。

关联

a) /member::NamedElement[*]——在命名空间中可标识的命名元素的汇集，这些元素或是属于命名空间的，或是通过引入或继承引进加的。这是一个派生联合。

b) /ownedMember::NamedElement[*]——由命名空间拥有的命名元素的汇集。Element::ownedElement 和 Namespace::member 的子集。这是一个派生的联合。

约束

[1] 命名空间的所有成员在内部是可区分的。

membersAreDistinguishable()

附加操作

[1] 查询 getNamesOfMember()给出了一个成员在命名空间中所具有的全部名称的集合。一般地，如果一个成员用不同的别名多次被引入，那么该成员可在该命名空间中有多个名称。那些语义可能通过重载 getNamesOfMember()操作来规定。这里的规约只是返回一个只包含单个名称的集合，如果没命名名称就返回空集。

```
Namespace::getNamesOfMember(element:NamedElement):Set(String);
getNamesOfMember=
if member->includes(element)then Set{}->including(element.name) else Set{} endif
```

[2] 布尔型查询 membersAreDistinguishable()确定命名空间的所有成员是否可在内部区分。

```
Namespace::memberAreDistinguishable():Boolean;
membersAreDistinguishable=
 self.member->forAll(memb|
      self.member->excluding(memb)->forAll(other|
          memb.isDistinguishableFrom(other,self)))
```

语义

命名空间为命名元素提供一个容器。它提供了一种分解复合名称(如名称 1::名称 2::名称 3)的方法。称为 N 的命名空间的成员都可以用 N::<x>形式的复合名称来引用，member 关联标识 N 中所有这种命名元素。注意不同于 N 中可引用为非限定的所有名称，因为该集合还包括所有包封装的命名空间的未隐藏的成员。

按照规定可区分规则命名元素如何同另一命名元素区分开来的规则，命名元素可出现在一个命名空间中。

缺省的规则是，如果两个元素有不相关的类型或者虽有相关的类型却有不同的名称，那么这两个元素就是可区分的。

在特别情况下，例如可由其特征标记区分的操作，这个规则可以被具体的子类重载。

记法

无附加的记法。具体的子类将自行定义特定的记法。

7.14 属权包(Ownershipspackage)

抽象包的属权子包将基本元素扩展以支持其他元素的属权。如图49、图50所示。

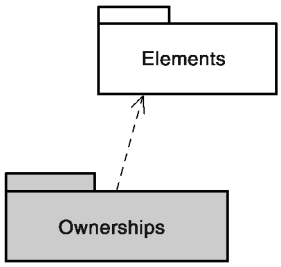


图49 属权包

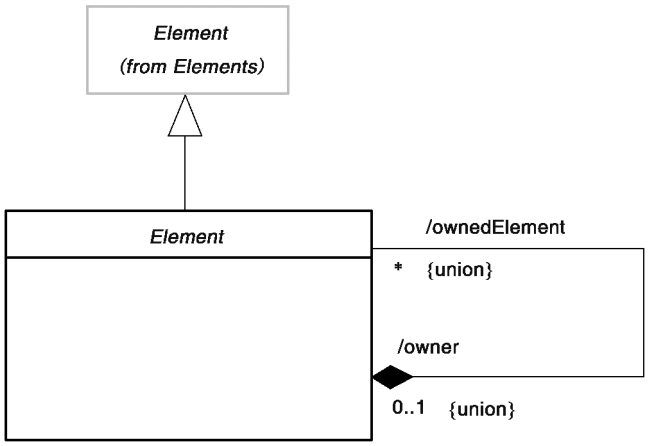


图50 属权包中定义的元素

7.14.1 元素[Element(特化的)]

元素是模型的成分。以此,它具有拥有其他元素的能力。

描述

元素Element有一个对于自身的派生复合关联,以支持一个元素拥有其他元素的通用能力。

属性

无附加属性。

关联

a) /ownedElement:Element[*]——Element由该元素所拥有。这是一个派生联合。

b) /owned:Element[0..1]——拥有该元素的Element。这是一个派生联合。

约束

[1] 一个元素不可直接或间接地拥有自身。

not self.allOwnedElements()->includes(self)

[2] 被拥有的元素应有一个属主。

self.mustBeOwned() **implies** owner->notEmpty()

附加操作

[1] 查询allOwnedElements()给出一个元素直接或间接拥有的所有元素。

Element::allOwnedElements():Set(Element);

allOwnedElements=ownedElement->union(ownedElement->collect(e|e.allOwnedElements()))

[2] 查询 mustBeOwned()指明该类型的元素是否应有一个属主。不需要属主的 Element 的子类应重载该操作。

Element::mustBeOwned():Boolean;

mustBeOwned=true

语义

元素 Element 的子类将为所表示的概念提供合适的语义。

派生的 ownedElement 关联是由元模型中所有组合关联端的(直接或者间接)子集。ownedElement 为访问直接为 Element 所拥有的元素提供了一种方便的方法。

记法

对元素 Element 没有通用记法。Element 的特定子类自行定义记法。

7.15 重定义包(Redefinitions package)

抽象包中的重定义包规定在泛化层次的语境中重定义的模型元素的通用能力。如图 51、图 52 所示。

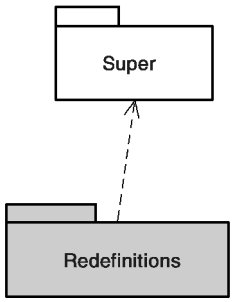


图 51 重定义包

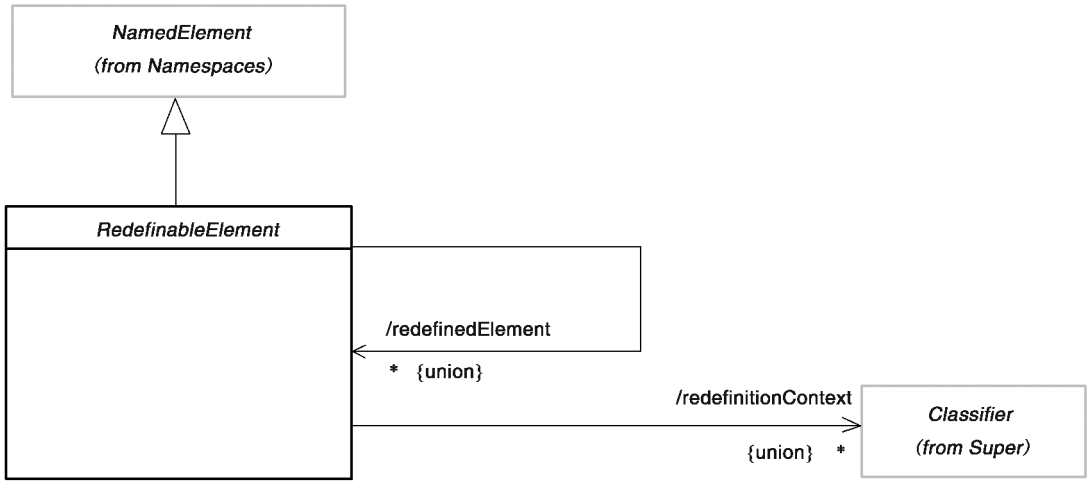


图 52 在重定义包中定义的元素

7.15.1 可重定义元素(RedefinableElement)

可重定义的元素是这样一种元素,在已经在类目的语境中定义时,还可在另一个类目的语境中进行更具体地或以不同方式重定义,这另一个类目将语境类目(直接或间接)特化。

描述

可重定义的元素是一种可在泛化语境中重定义的命名元素。RedefinableElement 是一个抽象元类。

属性

无附加属性。

关联

a) /redefinedElement:RedefinableElement[*]——作为正由该元素重定义的可重定义元素。这是一个派生联合。

b) /redefinitionContext:Classifier[*]——引用该元素可从中重定义的语境。这是一个派生的联合。

约束

[1] 重定义元素的各重定义语境至少有一个应是被重定义的元素各重定义的语境中至少一个的特化。

self.redefinedElement->forAll(e|self.isRedefinitionContextValid(e))

[2] 重定义元素应与每一个被重定义的元素一致。

self.redefinedElement->forAll(re|re.isConsistentWith(self))

附加操作

[1] 查询 isConsistentWith()对于其中有可能重定义语境中的任何两个 RedefinableElements,规定重定义在逻辑上是否一致。该操作的缺省值为假;对定义一致性条件的 RedefinableElement 的子类,应将这一操作重载。

RedefinableElement::isConsistentWith(redefinee:RedefinableElement):Boolean;

pre:redefinee.isRedefinitionContextValid(self)

isConsistentWith=false

[2] 查询 isRedefinitionContextValid()规定该 RedefinableElement 的重定义语境是否与规定的 RedefinableElement 的重定义语境正确相关,以使该元素能重定义其他元素。缺省情况是,这一元素的各重定义语境至少有一个应是所规定元素的各重定义的语境至少一个的特化。

RedefinableElement::isRedefinitionContexValid(redefinable:RedefinableElement):Boolean;

isRedefinitionContextValid=self.redefinitionContext->exists(c|

redefinable.redefinitionContext->exists(c|c.allParents()->includes(r)))

语义

RedefinableElement 在泛化联系的语境中表示被重定义的一般能力。重定义的详细语义随 RedefinableElement 每一次特化而变化。

可重定义元素是一个关于类目实例的规约,其中类目是该元素的重定义语境之一。对于将更一般的类目(直接地或间接地)特化的类目,可由另一元素重新定义一般类目中的元素,以便增加、约束或重载此规约,因为它更具体地施加到特化的类目的实例。

重定义元素应与它所重定义的元素一致,但可增加特定的约束,或者增加对特定重定义语境的实例的特别细节,该语境不与一般语境中的不变约束发生矛盾。

可重定义元素可以被重复定义多次。更有甚者,一个重定义中的元素可以重定义多重继承的可重定义元素。

语义变化点

被重定义的元素与重定义元素之间存在不同程度的兼容性,例如名称兼容性(重定义元素和被重定义元素的名称相同)、结构兼容性(被重定义元素的客户可见的性质同样也是重定义元素的性质),或者行为兼容性(重定义元素可以替代被重定义的元素)。任何一种兼容性都包括重定义的约束。特殊的选出语义是一个语义变化点。

记法

没有通用记法。参看 RedefinableElement 的子类所用的特定记法。

7.16 关系包(Relationships)

抽象包中的关系子包添加了对有向关系的支持。如图 53、图 54 所示。

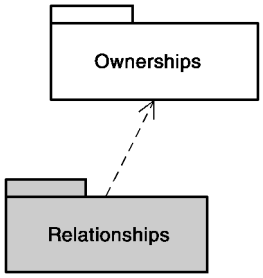


图 53 关系包

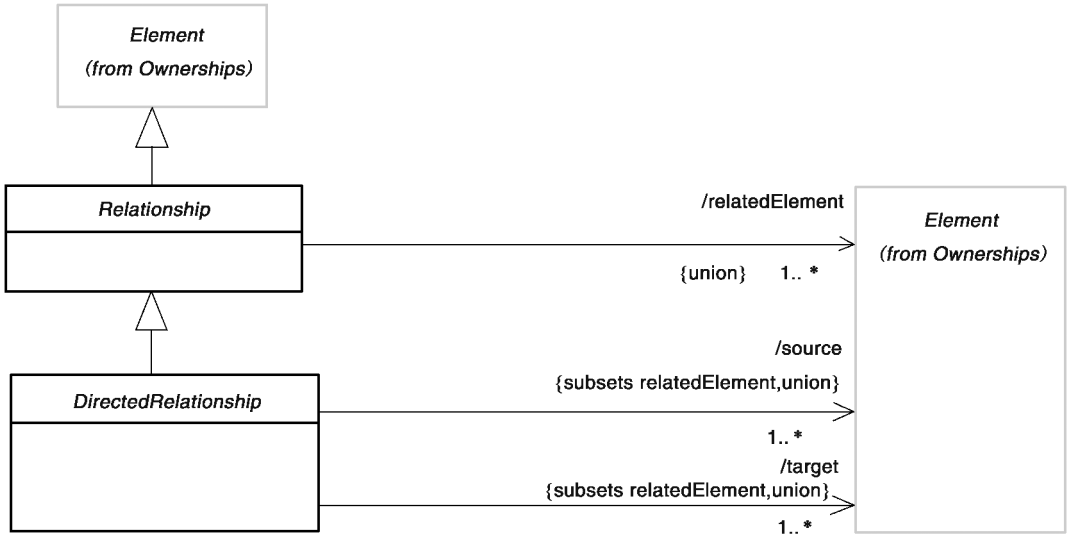


图 54 关系包中定义的元素

7.16.1 有向关系(DirectedRelationship)

有向关系表示源模型元素的汇集与目标模型元素的汇集之间的关系。

描述

有向关系引用一个或多个源元素和一个或多个目标元素。有向关系是一个抽象元类。

属性

无附加属性。

关联

a) /source:Element[1.. *]规定 DirectedRelationship 的资源。Relationship::relatedElement 的子集。这是一个派生联合。

b) /target:Element[1.. *]规定 DirectedRelationship 的目标。Relationship::relatedElement 的子集。这是一个派生联合。

约束

无附加约束。

语义

DirectedRelationship 无特定语义。DirectedRelationship 的各种子类将为它们所要表示的概念增

加适当的语义。

记法

DirectedRelationship 没有通用的记法。DirectedRelationship 的特定子类自行定义记法。大多数情况下，此记法是从源到目标之间画出线段的变体。

7.16.2 关系(Relationship)

关系是一种规定元素之间各种关系的抽象概念。

描述

关系引用一个或多个相关的元素。关系是一种抽象元类。

属性

无附加属性。

关联

/relatedElement:Element[1..*]——规定由联系所关联的元素。这是一个派生联合。

约束

无附加约束。

语义

关系无特定语义。Relationship 的各种子类将为它们所要表示的概念增加适当的语义。

记法

Relationship 没有通用的记法。Relationship 的特定子类将自行定义记法。大多数情况下，该记法是画在相关元素之间的连线的变体。

7.17 结构特征包(StructuralFeatures package)

抽象包的结构特征包规定了类目的结构特征的抽象泛化。如图 55、图 56 所示。

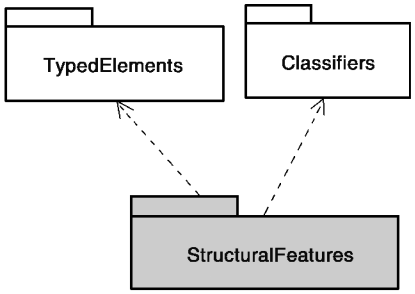


图 55 结构特征包

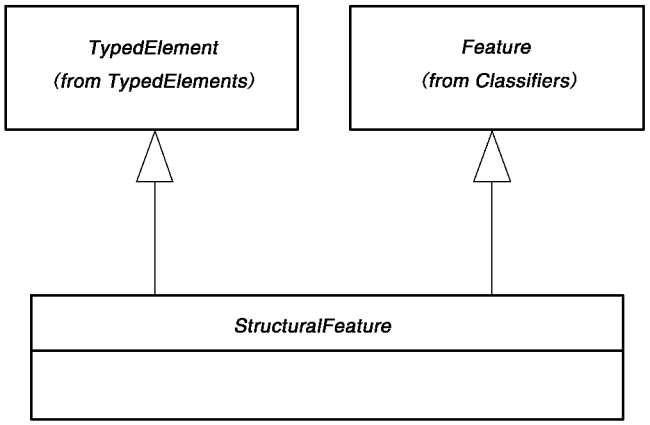


图 56 结构特征包中定义的元素

7.17.1 结构特征(StructuralFeature)

结构特征是一种规定类目实例的结构的类目的类型化的特征。

描述

结构特征是类目的类型化特征,它规定该类目实例的结构。结构特征是一个抽象元类。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

结构特征规定:将类目特征化的实例有一个其值具有所规定类型的槽。

记法

无附加的记法。

7.18 超包(Super package)

抽象包的超包提供规定类目之间的泛化关系的机制。如图 57、图 58 所示。

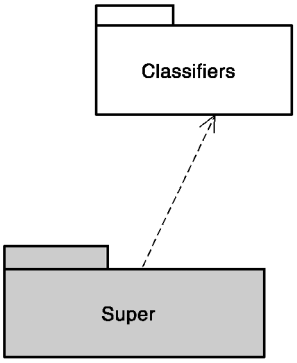


图 57 超包

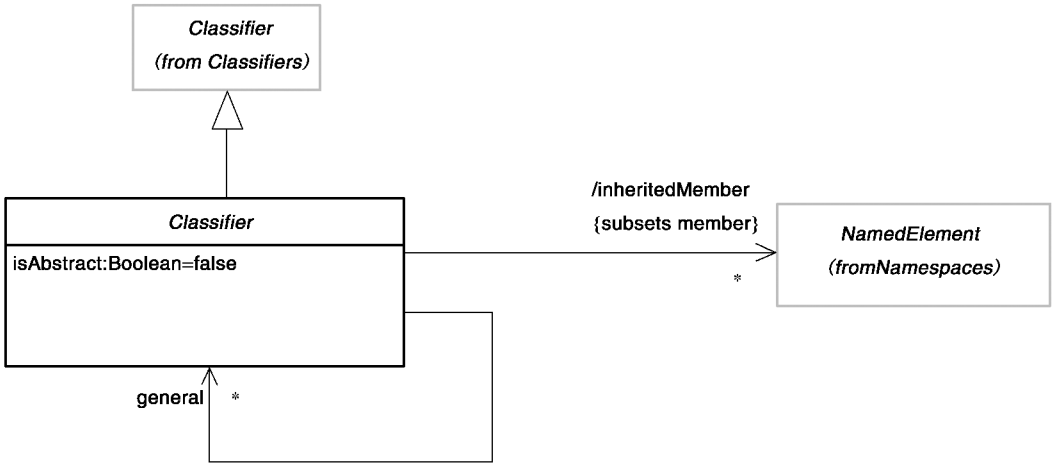


图 58 超包中定义的元素

7.18.1 类目 Classifier(特化的)

描述

类目可以通过引用其通用类目来规定泛化层次。

属性

isAbstract:Boolean——如果为真,类目就不提供完整的声明,一般也不能被实例化。抽象类目是供其他类目使用,即作为通用元关系(metarelationships)或者泛化关系的目标。

关联

a) general:Classifier[*]——规定这一类目的泛化层次中更通用的类目;

b) /inheritedMember: NamedElement[*]——规定该类目从一般类目所继承的所有元素。Namespace::member 的子集。这是一个派生联合。

约束

[1] 泛化层次应是有向的和非循环的。类目不允许是同一类目的传递的通用类目,也不允许是传递的特定类目。

not self.allParents()->includes(self)

[2] 类目只可特化有效类型的类目。

Self.parents()->forAll(c|self.maySpecializaType(c))

[3] inheritedMember 关联由继承其父代的可继承的成员派生而来。

self.inheritedMember-> includesAll (self.inherit (self.parents ()-> collect (p | p.inheritableMembers(self))))

附加操作

[1] 查询 parents()给出泛化类目的所有直接祖先。

Classifier::parents():Set(Classifier);

parents=general

[2] 查询 allParents()给出泛化类目的所有直接的或间接的祖先。

Classifier::allParents():Set(Classifier);

allParent=self.parents()->union(self.parents()->collect(p|p.allParents()))

[3] 查询 inheritableMembers()给出一个类目可被其后辈之一继承的所有成员,受可见度的限制。

Classifier::inheritableMembers(c:Classifier):Set(NamedElement);

pre:c.allParents()->includes(self)

inheritableMembers=member->select(m|c.hasVisibilityOf(m))

[4] 查询 VisibilityOf()确定类目中的命名元素是否可见。缺省情况是所有元素都可见。该操作只有在该变元被一父代拥有时才会被调用。

Classifier::hasVisibilityOf(n:NamedElement):Boolean;

pre:self.allParents()->collect(c|c.member)->includes(n)

hasVisibilityOf=true

[5] 查询 inherit()定义如何继承元素集合。这里,定义该操作就是为了继承所有元素。当继承受到重定义影响时,该操作将予以重定义。

Classifier::inherit(inhs:Set(NamedElement)):Set(NamedElement);

inherit=inhs

[6] 查询 maySpecializaType()确定该类目是否会与特定类型的类目形成泛化关系。缺省情况为一个类目可特化具有相同类型或更通用类型的类目。它由具有不同特化约束的类目来重定义。

Classifier::maySpecializeType(c:Classifier):Boolean;

maySpecializeType＝self. oclIsKindOf(c. oclType)

语义

泛化对类目的具体子类型的特定语义有不同影响。

特定类目的实例也是每一个通用类目的(直接)实例。因此,通用类目实例所规定的特征对特定类目实例以隐式规定。作用于通用类目实例的约束同样也作用于特定类目的实例。

记法

抽象类目的名称以斜体字示出。

泛化通过一条带空心三角箭头的线段图示,连接两个表示所涉及的类目的符号。箭头指向表示通用类目的符号。这种记法称为“分离目标样式”。参看下面的例子。

表示选项

具有相同通用类目的多个类目可与“共享目标样式”一起图示。参看以下例子。

抽象类目可在类目名称的后面或下面以关键字{abstract}来图示。如图 59 所示。

示例

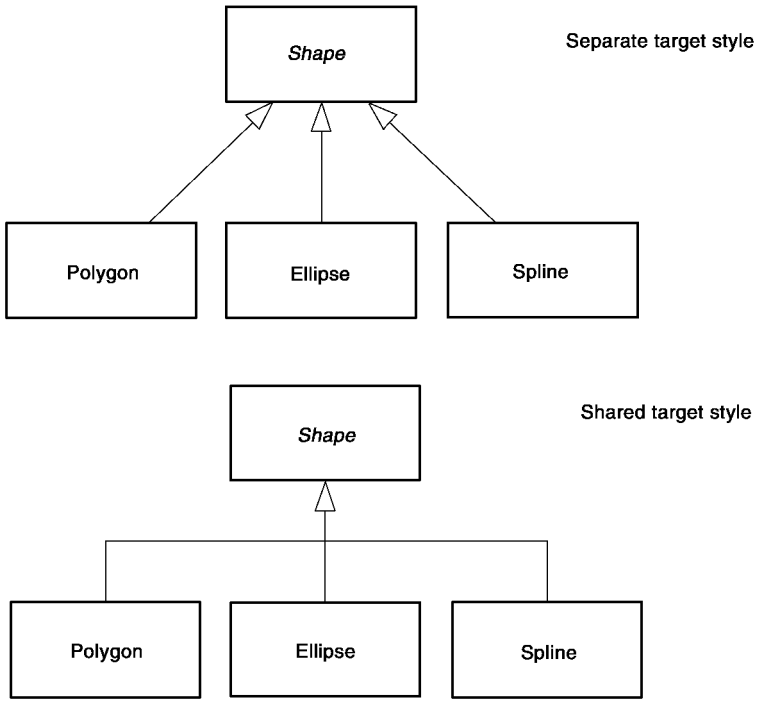


图 59　泛化层次中类的例子

7.19　类型化元素包(TypedElements)

抽象包的类型元素 TypedElement 子包定义类型化的元素及其类型。如图 60、图 61 所示。

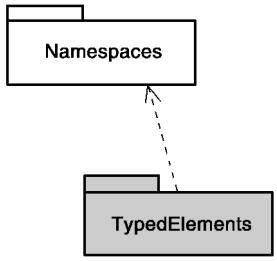


图 60　类型化元素包

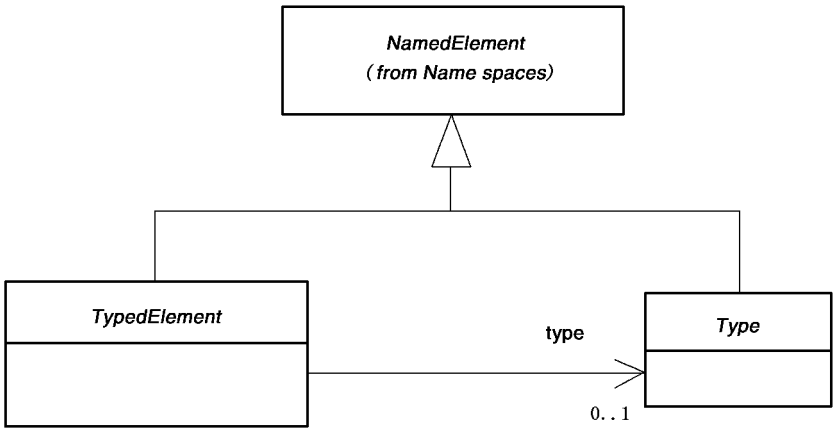


图 61　类型元素包中定义的元素

7.19.1　类型(Type)

类型约束了类型元素所表示的值。

描述

类型用作对由类型化元素所表示的值的范围的约束。类型 Type 是一个抽象元类。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

附加操作

[1]　查询 conformsTo()给出对与另一类型兼容的类型为真的值。缺省情况下两个类型互不一致。在特定一致性的情况下,该查询需加以重定义。

conformsTo (other:Type):Boolean;

conformsTo=false

语义

类型表示一个值集。对具有该类型的类型化元素加以约束,以便表示这个值集的值。

记法

无附加记法。

7.19.2　类型元素(TypedElement)

每一个类型元素都有类型。

描述

类型元素是一种具有作为对该元素所表示的值的约束的某种类型的元素。类型化元素是一种抽象元类。

属性

无附加属性。

关联

type:Type[0..1]——类型化元素的类型。

约束

无附加约束。

语义

对由该元素所表示的值加以约束,使之成为此类型的实例。没有关联类型的类型化元素可以表示任何类型的值。

记法

没有通用的记法。

7.20 可见性包(Visibilities)

7.20.0 概述

抽象包的可见性包提供了从其构建可见性语义的基本构造。如图 62、图 63 所示。

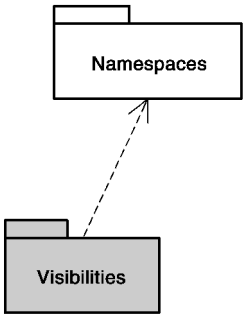


图 62 可见性包

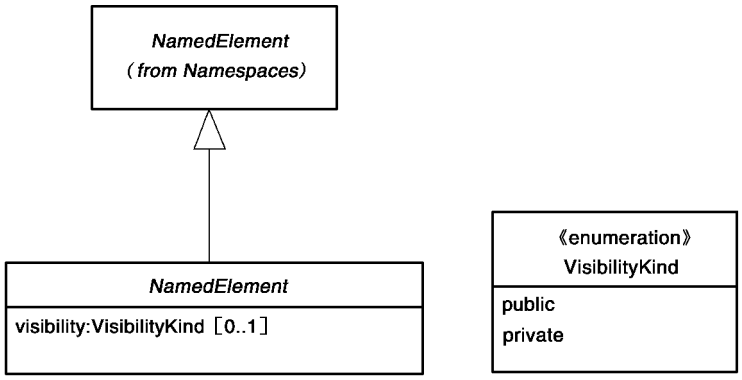


图 63 可见性包中定义的元素

7.20.1 命名元素[NamedElement](特化的)

描述

命名元素 NamedElement 有一个可见性属性。

属性

visibility：VisibilityKind[0..1]——确定在总体模型内的不同命名空间里的 NamedElement 的可见性。

关联

无附加关联。

约束

[1] 当一个 NamedElement 不属于任何一个命名空间时，它就没有可见性。

namespace->isEmpty () **implies** visibility->isEmpty ()

语义

可见性属性提供了对模型中不同命名空间的命名元素的用法进行约束的方法。该属性与引入和泛化机制结合使用。

7.20.2 可见性种类(VisibilityKind)

可见性种类 VisibilityKind 是一种定义确定模型中元素的可见性的文字的枚举类型。

描述

VisibilityKind 是一种有以下枚举值的枚举：

a) 公有的；

b) 私有的。

附加操作

[1] 查询 bestVisibility()检查 VisibilityKind 的集合，并返回 public 作为首选的可见性。

VisibilityKind::bestVisibility(vis：Set(VisibilityKind))：VisibilityKind；

bestVisibility=**if** vis->includes(#public)**then** #public **else** #private **endif**

语义

VisibilityKind 与比如 Imports、Gen-eralizations 和 Packages 包一起，用于在关联中的可见性规约。详细语义与这些机制结合起来规定。如果 Visibility 包不与这些包结合起来使用，那么这些文字就会有不同的意义，或者没有意义。

a) 公有元素对所有可访问所属命名空间内容的元素可见。

b) 私有元素只在所属的命名空间内可见。

在命名元素最终具有多重可见性(比如通过被多次引入)的情形，公有可见性将覆盖私有可见性，也就是说，当将一个元素两次引入同一个命名空间时，一次采用公有引入，一次是私有引入，那么它是公有的。

8 核心::基本的(Core::Basic)

8.0 概述

InfrastructureLibrary::Core 的基本 Basic 包提供了一种基于类的最小建模语言，可在其上建立更为复杂的语言。元对象设施(Meta-Object Facility)的实质层(Essential Layer)用基本包来实现重用。基本包中的元类采用以下四种图来规定：类型图(Types)、类图(Classes)、数据类型图(DataTypes)和包图(Packages)。可将基本包视为它自己的一个实例。构造(Construct)包中定义的基本包构造要更为复杂，元对象设施的完全层(Complete layer)和 UML 超结构(Superstructure)利用该构造包来实现重用。如图 64 所示。

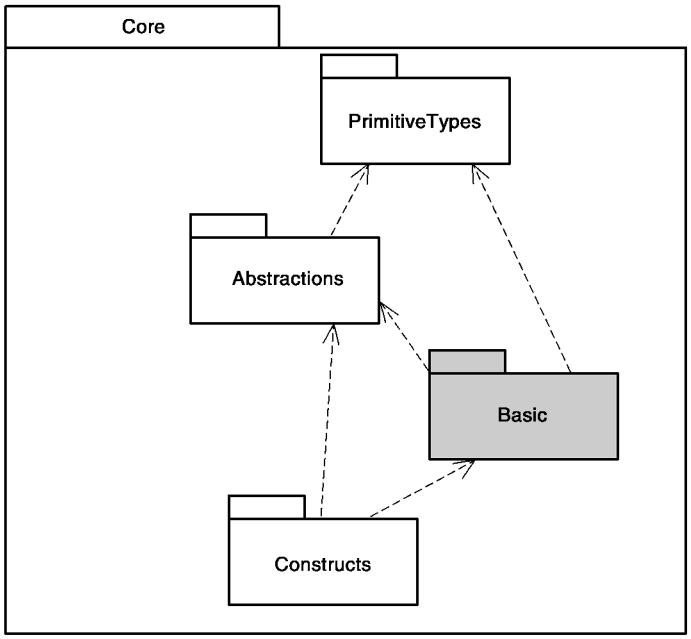


图 64　核心包属于基础结构库包(并包含有多种子包)

8.1　类型图(Type diagram)

类型图定义了抽象元类,抽象元类对元素进行命名和类型定义。如图 65 所示。

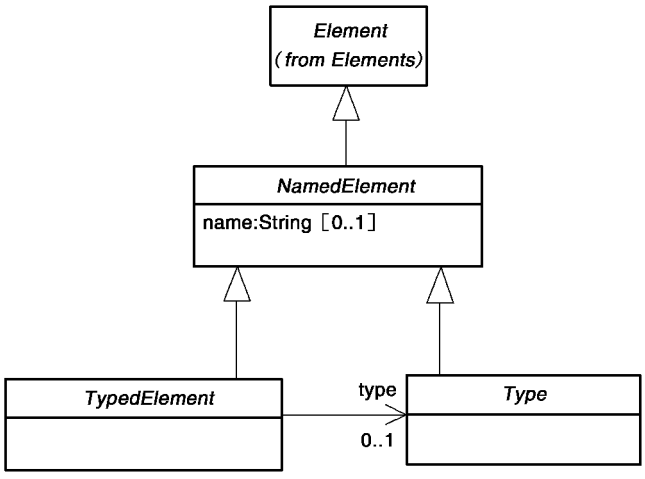


图 65　类型图中定义的类

8.1.1 类型(Type)

描述

类型是一种用作类型化元素的类型的已命名元素。

属性

无附加属性。

语义

类型是一种规定类型化元素的类型的一般概念,并约束该类型化元素可引用的值的集合抽象类。

记法

Basic::Type 作为抽象类没有记法。

8.1.2 命名元素(NamedElement)

描述

命名元素表示具有名称的元素。

属性

name:String[0..1]. ——元素的名称。

语义

具有名称的元素是命名元素 NamedElement 的实例。已命名元素的名称是可选的。如果加以规定,则可使用任何有效的串,包括空串。

记法

Basic:NamedElement——作为一个抽象类没有记法。

8.1.3 类型化元素(TypedElement)

描述

类型元素也是一种表示具有类型的元素的命名元素。

属性

type:Type[0..1]——该元素的类型。

语义

具有类型的元素是 TypedElement 的一个实例。类型化元素可以可选地没有类型。类型元素的类型约束了该类型化元素可引用的值的集合。

记法

Basic::TypeElement——作为抽象类没有记法。

8.2 类图(Classes diagram)

类图为基于类的建模定义了构造。如图 66 所示。

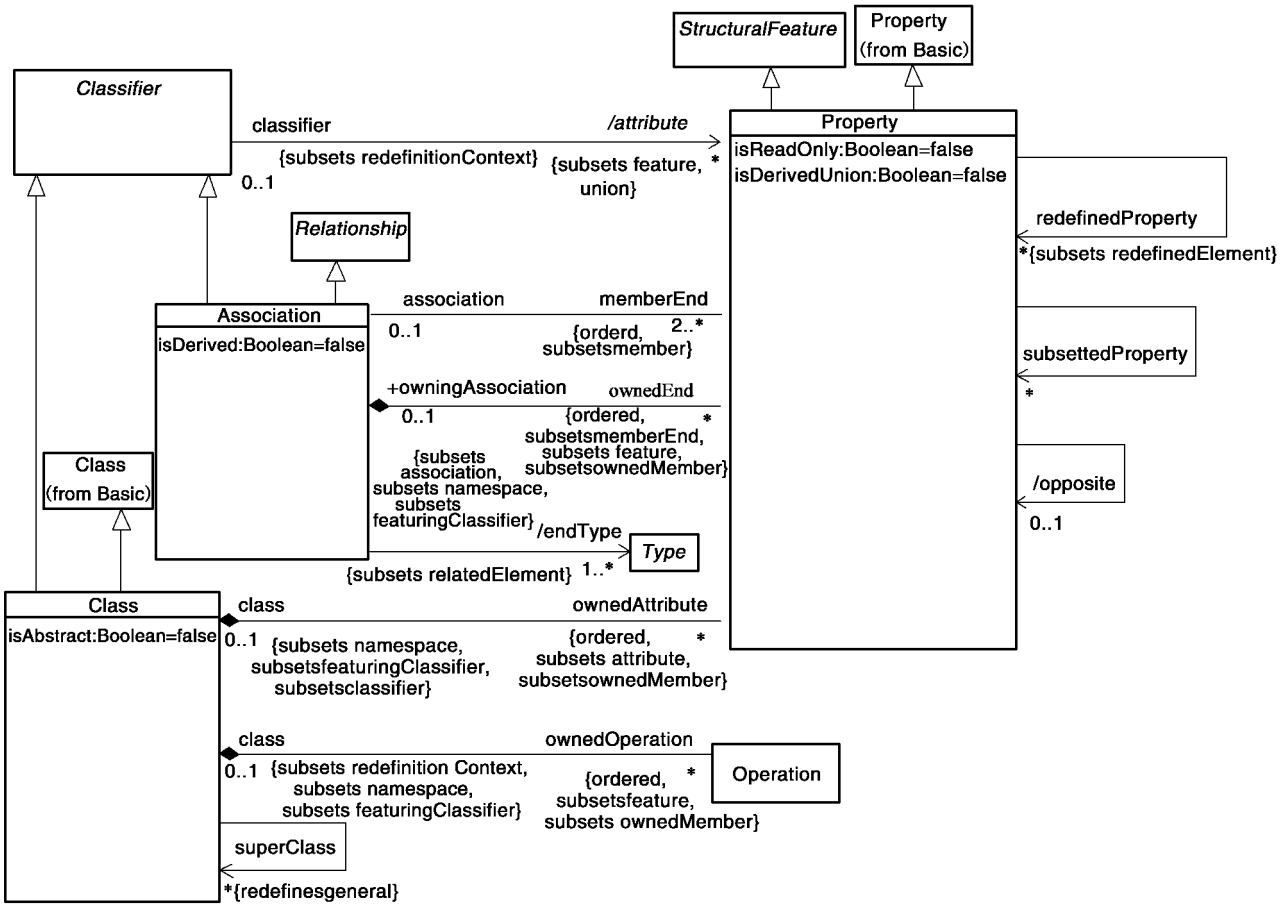


图 66　类图中定义的类

8.2.1　类(Class)

描述

类是一种把对象作为其实例的类型。

属性

a)　isAbstract:Boolean——当类是抽象类时为真。缺省值为假。

b)　ownedAttribute:Property[*]——类拥有的属性。这些属性不包括继承来的属性。属性由性质(Property)的实例来表示。

c)　ownedOperation:Operation[*]——类拥有的操作。不包括继承来的操作。

d)　superClass:Class[*]——类从中继承的类的直接超类。

语义

类有属性和操作,并参与继承层次结构。允许多重继承。类的实例就是对象。如果一个类为抽象类,它就没有任何直接实例。任何具体的(即非抽象的)类的实例也都是其类的超类的直接实例。对象为其类的每个直接的和继承的属性都有一槽。对象允许调用它的类和类的超类中定义的操作。这种调用的语境就是被调用的对象。

记法

Basic::Class 的记法和 Constructs::Class 的一样,省略了记法中不能用 Basic 模型表示的部分。

8.2.2 操作(Operation)

描述

操作属于类,并可在作为该类的对象的语境中调用。这是一种类型化元素和势域元素。

属性

a) class:Class[0..1]——拥有该操作的类;

b) ownedParameter:Parameter[*]{ordered,composite}——操作的参数;

c) RaisedException:Type[*]——调用该操作期间声明为可能的异常。

语义

操作属于某一个类。对该类的任何一个直接或间接实例的对象都可调用一操作。在这种调用中,执行语境包括这一对象和参数的值。操作的类型(如果有时)就是操作返回结果的类型,势域就是该结果的势域。操作可以和一组表示可能异常的类型相关联,这些异常可由操作引起。

记法

Basic::Class 的记法和 Constructs::Class 的一样,省略了那些记法中不能由 Basic 模型表示的部分。

8.2.3 参数(Parameter)

描述

参数是一种表示操作的参数的类型化元素。

属性

operation:Operation[0..1]——拥有参数的操作。

语义

当调用一个操作时,可对每一个参数赋值向其传递一个变元。每个参数都有类型和势域。每个 Basic::Parameter 都与一个操作相关联,尽管在 UML 模型中其他地方的参数的子类并不与一个操作相关联,因此势域为 0..1。

记法

Basic::Parameter 的记法和 Constructs::Parameter 的一样,但省略了记法中那些不能由 Basic 模型不收的部分。

8.2.4 性质(Property)

描述

性质是一种表示类的属性的类型化元素。

属性

a) class:Class[0..1]——拥有属性的类,且性质是其属性。

b) default:String[0..1]——当对所属类被实例化时,对其求值以给出属性的缺省值的串。

c) isComposite:Boolean——如果 isComposite 为真,则包含属性的对象是属性所含对象或值的容器。缺省值为假。

d) isDerived:Boolean——如果 isDerived 为真,则属性值是从其他处的信息导出。缺省值为假。

e) isReadOnly:Boolean——如果 IsReadOnly 为真,则初始化以后就不可写出属性。缺省值为假。

f) opposite:Property[0..1]——对象 o1 和 o2(可以是相同的对象)的属性 attr1 和 attr2 可以成对出现,这样 o1.attr1 当且仅当 o2.attr2 引用 o1 时引用 o2。在这样的情况下,attr1 和 attr2 互相对立。

语义

性质表示类的属性。性质具有类型和势域。当性质和它的相对成分配对时，它们表示两个互相约束的属性。互相对立的两个性质的语义，与 Constructs 中的双向导航的关联是相同的，但关联没有用作实例的显式链接和名称。

记法

当 Basic::Property 没有对立的性质时，它的记法就和 Constructs::Property 用作属性的一样，但省略了记法中不能由 Basic 模型表示的部分。正常情况下，如果性质的类型是数据类型，则属性在类框的属性格子内图示；如果性质的类型是类，那么它将采用类似关联的箭头记法来图示。

如果性质有对立对象，那么这对属性采用与 Constructs::Association 相同的记法来图示，带有两个可导航端，但省略了记法中不能用 Basic 模型表示的部分。

8.3 数据类型图(DataTypes diagram)

数据类型图定义了定义数据类型的元类。如图 67 所示。

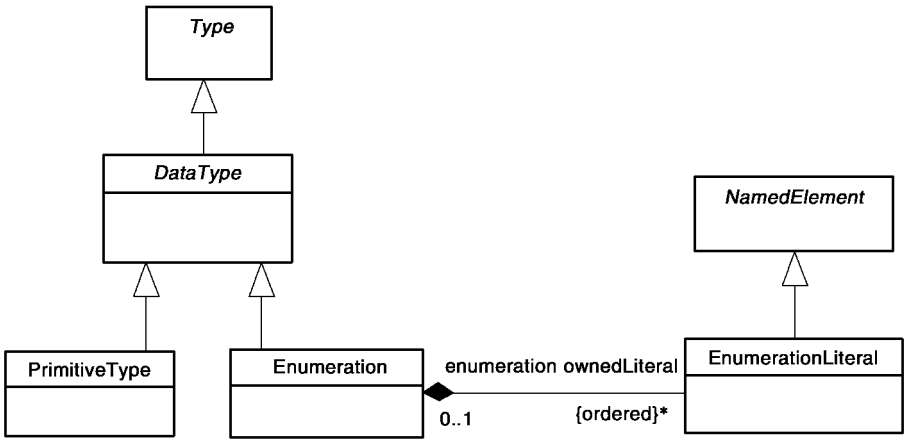


图 67 数据类型图中定义的类

8.3.1 数据类型(DataType)

描述

数据类型类是用作一种各种不同数据类型的共同超类的抽象类。

属性

无附加属性。

语义

DataType 是一种表示正作为数据类型(即其实例只能由这些实例的值来标识的类型)的通用概念的抽象类。

记法

Basic::DataType 作为一种抽象类没有记法。

8.3.2 枚举(Enumeration)

描述

枚举定义了一组可用作其值的文字。

属性

ownedLiteral:EnumerationLiteral[*]{ordered, composite}——枚举文字的有序汇集。

语义

枚举定义了一个有限的有序值集合,比如{red,green,blue}。尤其类型是枚举的类型化元素指代的值应取自这个集合。

记法

Basic::Enumeration 的记法和 Constructs::Enumeration 的一样,但省略了那些记法中不能由 Basic 模型表示的部分。

8.3.3 枚举文字(EnumerationLiteral)

枚举文字是枚举的一个值。

属性

enumeration:Enumeration[0..1]——这一文字所属的枚举。

语义

见“枚举类型”。

记法

见“枚举类型”。

8.3.4 原子类型(PrimitiveType)

描述

原子类型是一种由底层的基础设施实现并在建模可用的数据类型。

属性

无附加属性。

语义

在 Basic 模型本身用到的原子类型是 Integer、Boolean、String 和 UnlimitedNatural。它们的具体语义由所用工具的语境或者元模型的扩展(例如 OCL)给出。

记法

原子类型的记法与其实现有关。UML 元模型中使用的原子类型的记法在 Core::PrimitiveTypes 这一章中给出。

8.4 包图(Packages diagram)

包图定义了与 Package 及其内容有关的基本构造。如图 68 所示。

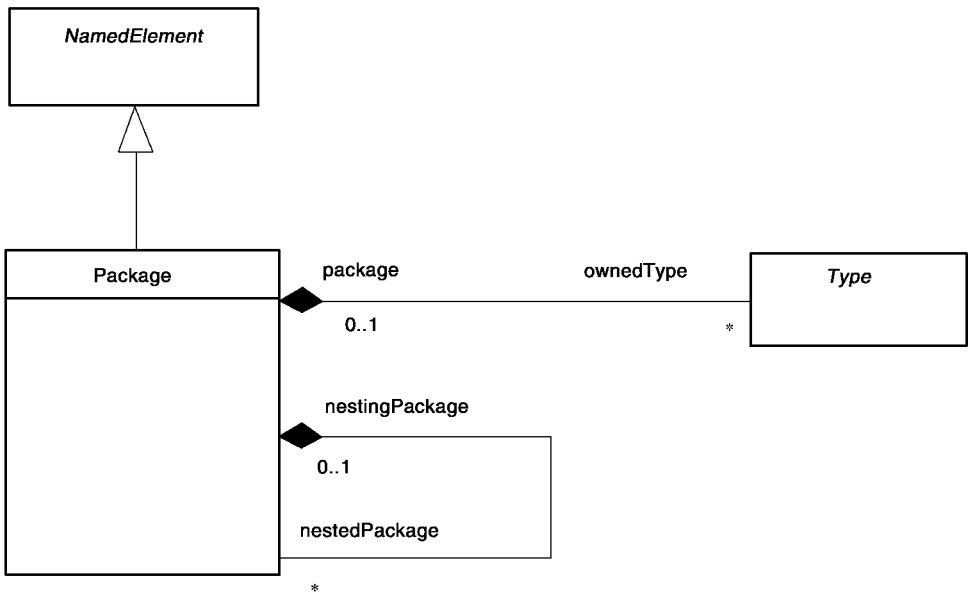


图 68　包图中定义的类

8.4.1　包(Package)

描述

包是类型和其他包的容器。

属性

a)　nestedpackage:Package[*]{composite}——一组被包含的包;

b)　nestingPackage:Package[0..1]——包含包;

c)　OwnedType:Type[*]{composite}——被包含类型的集合。

语义

包提供了一种将类型和包分组在一起的方式,这对理解和管理模型有用。包不能包含自身。

记法

将包和类型包含在包内所使用的记法和 Constructs::Package 的一样,但省略了记法中不能用 Basic 模型表示的部分。

8.4.2　类型[type(附加性质)]

描述

类型可以包含在一包中。

属性

package:Package[0..1]——包含包。

语义

无附加语义。

记法

包对类型的包含使用的记法和 Constructs::Package 的一样,它省略了记法中不能用 Basic 模型表示的部分。

9 核心::构造(Core::Constructs)

9.0 概述

本章描述了基础设施库::核心 InfrastructureLibrary::Core 中的构造包 Constructs。在元对象设施中拟重用构造包。如图 69 所示。

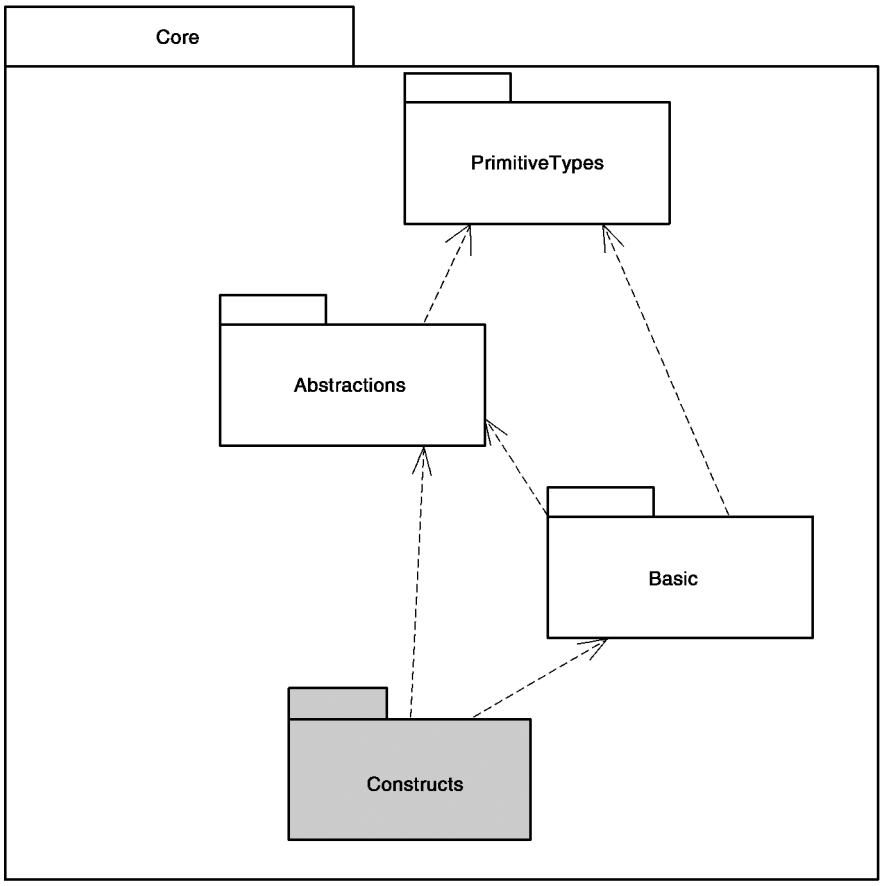


图 69 核心包为基础构造库所拥有并包含若干子包

构造包由下面各个章条中的一系列图规定。构造包依赖于若干其他的包,特别是基本包和抽象包中的各种包,如图 70 所示。

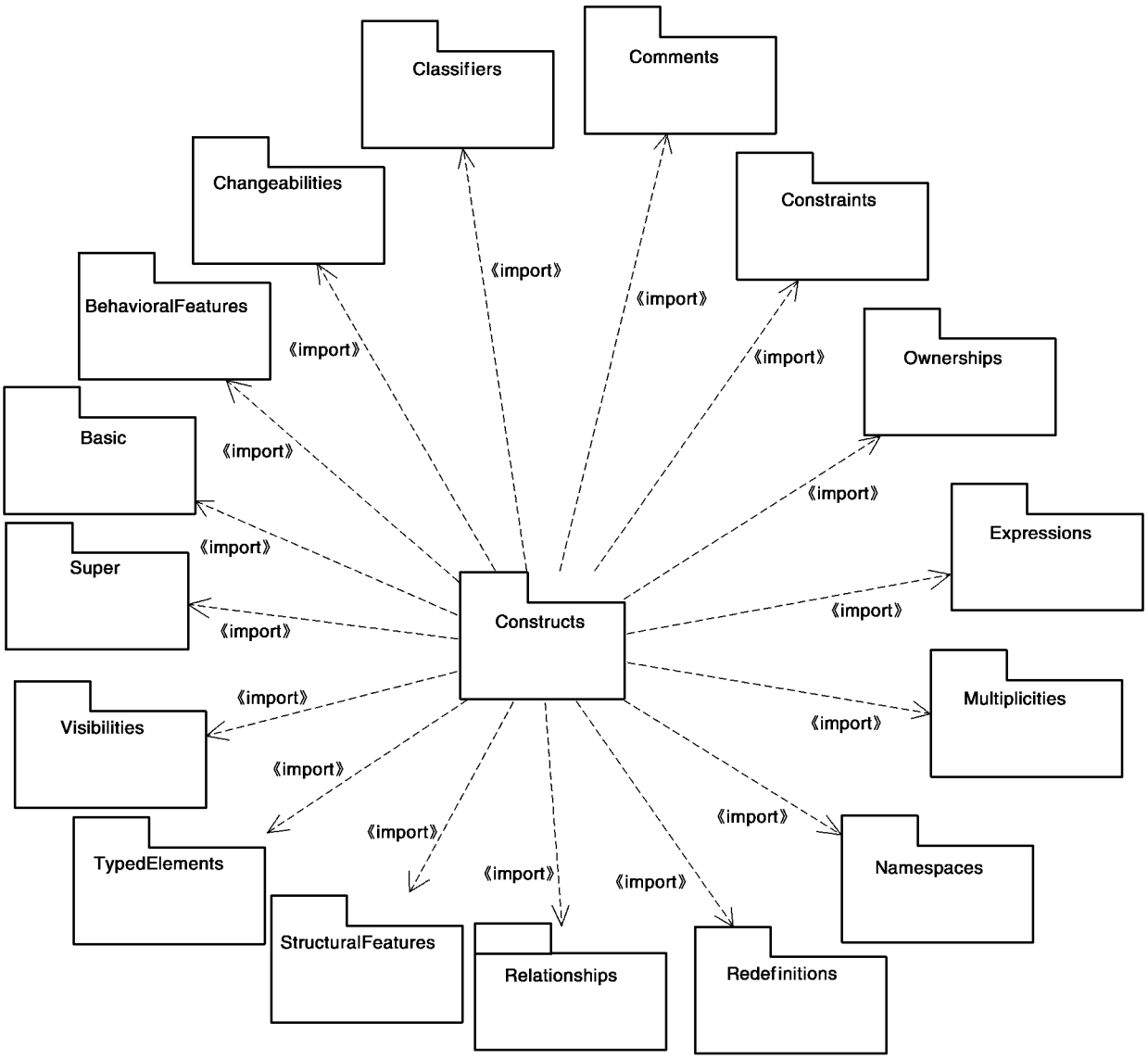


图 70　构造包依赖于若干其他包

9.1　根图（RootDiagram）

构造包中的根图规定元素、联系、有向联系和注释构造。如图 71 所示。

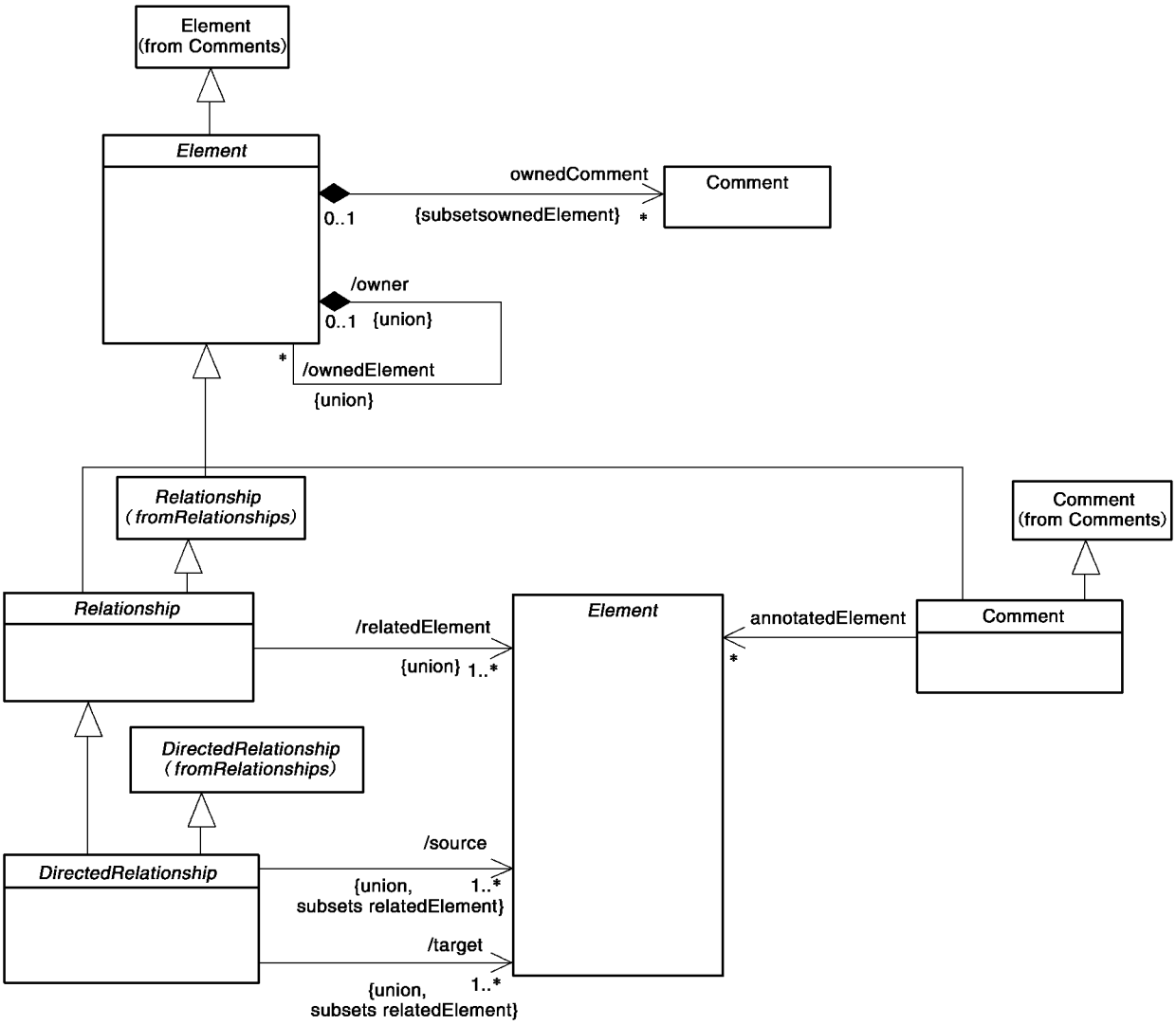


图 71　构造包中的根图

9.1.1　注释[Comment(特化的)]

描述

Constructs::Comment 重用了 Abstractions::Comments 中注释的定义。并增加了对 Constructs::Element 的特化。

属性

无附加属性。

关联

annotatedElement:Element[*]重定义抽象包中的相应性质。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.1.2 有向关系[DirectedRelationships(特化的)]

描述

Constructs::DirectedRelationship 重用 Abstractions::Relationships 中有向关系的定义。它增加了对 Constructs::Relationship 的特化。

属性

无附加属性。

关联

a) /source:Element[1..*]——重定义抽象包中相应的性质。Relationship::relatedElement 的子集。它是一个派生联合。

b) /target:Element[1..*]——重定义抽象包中相应的性质。Relationship::relatedElement 的子集。它是一个派生联合。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.1.3 元素[Element(特化的)]

描述

Constructs::Element 重用 Abstractions::Comments 中元素的定义。

属性

无附加属性。

关联

a) ownedComment:Comment[*]——重定义抽象包中相应的性质。

b) Element::ownedElement 的子集。

c) /ownedElement:Element[*]——重定义抽象包中相应的性质。它是一个派生联合。

d) /owner:Element[0..1]——重定义抽象包中相应的性质。它是一个派生联合。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.1.4 关系[Relationship(特化的)]

描述

Constructs::Relationship——重用 Abstractions::Relationship 中关系的定义,并对 Constructs::Element 增加了特化。

属性

无附加属性。

关联

/relatedElement:Element[1..*]——重定义抽象包中相应的性质。它是一个派生联合。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.2 表达式图(Expressions diagram)

构造包中的表达式图规定值规约、表达式和不透明表达式构造。如图 72 所示。

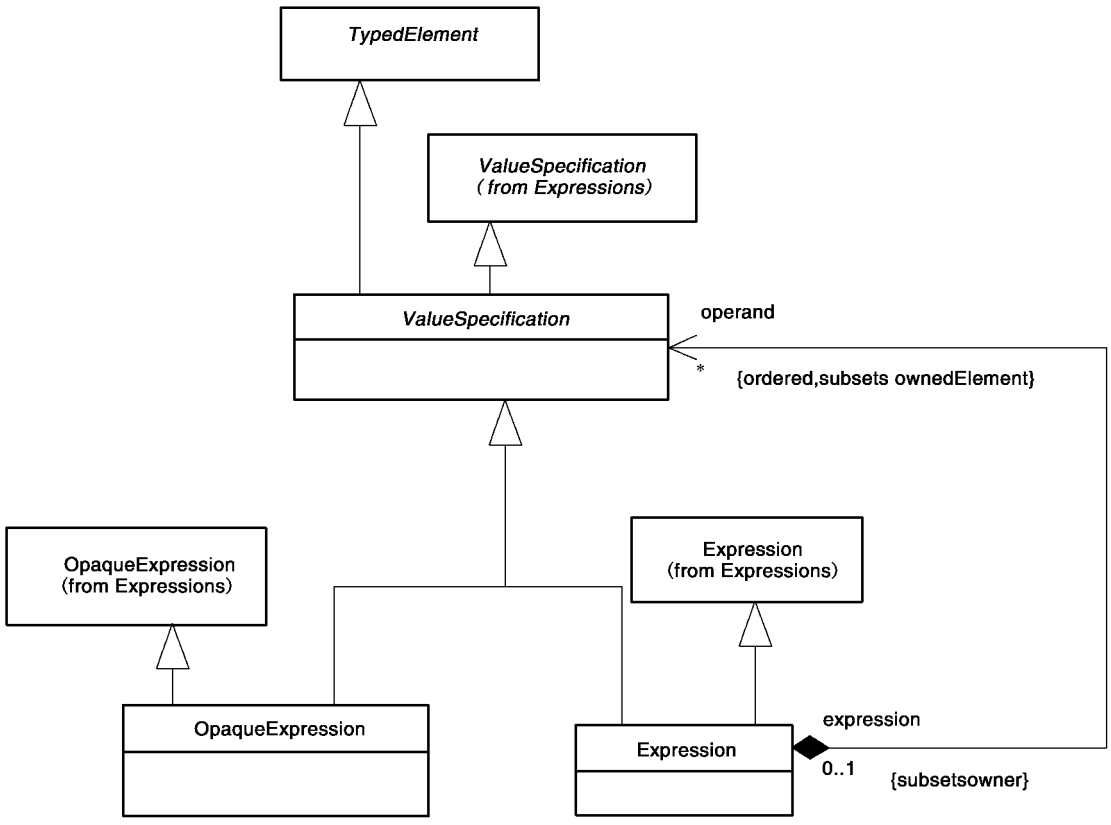


图 72 构造包中的表达式图

9.2.1 表达式[Expression(特化的)]

描述

Constructs::Expression——重用 Abstractions::Expression 中表达式的定义,并对 Constructs::ValueSpecification 增加了特化。

属性

无附加属性。

关联

/relatedElement:Element[1..*]——重定义抽象包中相应的性质。它是一个派生联合。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.2.2 不透明表达式[OpaqueExpression(特化的)]

描述

Constructs::OpaqueExpression——重用 Abstractions::OpaqueExpression. 中不透明表达式的定义,并对 Constructs::ValueSpecification 增加了特化。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.2.3 值规约[ValueSpecification(特化的)]

描述

Constructs::ValueSpecification 重用 Abstractions::ValueSpecification. 中值规约的定义,并对 Constructs::TypedElement 增加了特化。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.3 类图(Classes diagram)

构造包中的类图规定关联、类和性质的构造,并增加了类目和操作构造的特征。如图 73 所示。

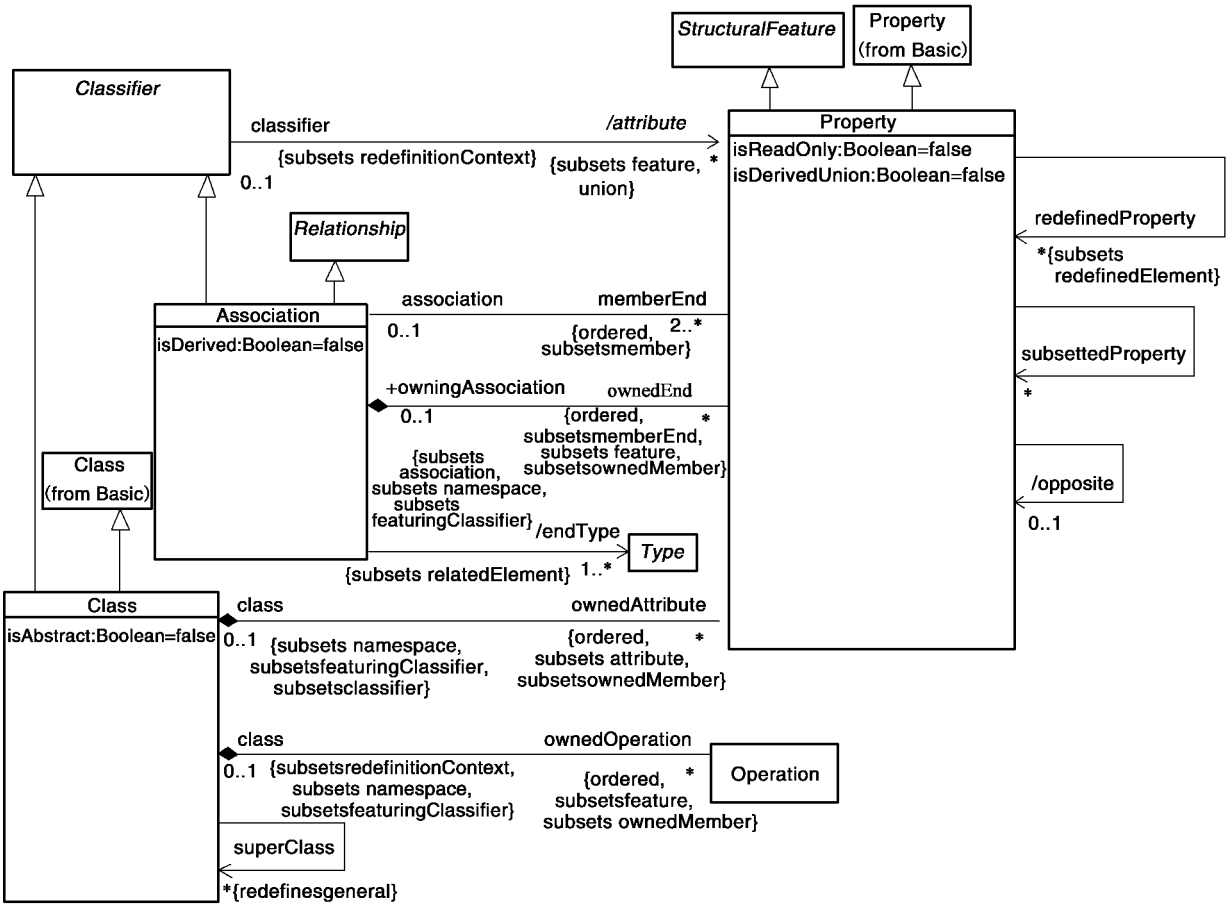


图 73　构造包中的类图

9.3.1　关联(Association)

关联描述一组其值引用类型化实例的元组。关联的实例叫做链接。

描述

关联定义了可在两个类型化实例之间出现的语义联系。它至少有两个由性质来表示的端,各连接到该端的类型。关联的多个端点可能有同一类型。

当性质属于关联时,它表示该关联的不可导航端。在这种情况下,性质不出现在任何相关联类目的命名空间中。当在关联的某端的性质属于一个相关联的类目时,它表示该关联的一个可导航的端。在这种情况下,该性质也是相关联类目的一个属性。只有二元关联可以有可导航端。

属性

isDerived:Boolean——规定该关联是否是由另一模型元素(例如其他关联或者约束)派生而来。缺省值是 false。

关联

a) memberEnd:Property[2..*]——每一端都表示类目实例的参与,这些类目连接到关联链接的端。它是有序关联。Namespace::member 的子集。

b) ownedEnd:Property[*]——关联本身所拥有的非导航端。它是有序关联。是:Association::memberEnd,Classifier::feature 和 Namespace::ownedMember 的子集。

c) /endType:Type[1..*]——引用作为关联端的类型的类目。

约束

[1] 将另一关联特化的关联具有和另一关联相同的端数。

self.parents()->forAll(p|p.memberEnd.size()=self.memberEnd.size())

[2] 当一个关联特化另一关联时，每个特定关联的端点都对应于其普遍关联的一端，且到达更普遍端的相同的类型或子类型。

[3] 端类型由成员端的类型派生而来。

self.endType=self.memberEnd->collect(e|e.type)

语义

关联声明被关联类型的实例之间可以存在链接。链接是一个对关联的每一端都带有一个值的，其中每个值都是该端的类型的一个实例。

当关联的一个或多个端有 isUnique=false 时，则可能有多个链接关联到相同的实例集上。这种情况下，链接带有一个附加的标识符以区分它们的端值。

当关联的一个或多个端有序时，链接除端值之外还带有定序信息。

对于有 N 个端的关联，选择其中任意 $N-1$ 个端及与这些端关联的特定实例。引用这些特定实例的关联的链接的汇集将标识在另一端的实例的汇集。关联端的势域约束这一汇集的大小。如果此端标注为有序的，那么这一汇集也有序。如果此端标注为惟一的，那么这一汇集是一个集合；否则允许有重复元素。

在某些条件下，关联的端可以标注为另外一关联的端的子集。这些条件是：

a) 两者有相同的端数。

b) 子集中关联所链接的每个类型集都与被子集的关联所连接的每个类型相一致。在这种情况下，考虑两种关联各自的另一端的特定实例的集合，则由子集端所指代的汇集完全包含在由被子集划分的端所指代的汇集中。

在某些条件下，关联的端可以标注为重定义另一关联的端。这些条件是：

a) 两者有相同的端数。

b) 重定义关联所连接的每个类型集都与被重定义的关联所连接的对应类型相一致。在这种情况下，考虑两种关联各自的另一端的特定实例的集合，则由重定义端所指代的汇集与被重定义端所指代的汇集相同。

关联可以被特化。特化关联的链接的存在隐含着与被特化关联中同一实例集合的链接的存在。

可导航的关联端与被关联实例的属性具有相同的语义。

注：对于 n 维关联，一端的低势域通常为 0。如果 n 维关联的一个端的势域为 1(或更多)，则隐含：对另一端的每一可能组合都应存在一个(或更多)链接。

关联可以表示一个组合聚合(即整体与部分联系)。只有二元关联可以是聚合。组合聚合是聚合要求部分实例在某一时刻最多只能被包含在一个组合内的强形式。如果组合被删除，则其所有部分通常也将随之被删除。注意一部分可以在(在允许处)组合被删除之前从复合中除去，因此不作为该组合的部分进行删除。组合定义传递性不对称关系——它们的链接形成一个有向的非循环图。在正被设置为 True 的关联的部分端，由 isComposite 属性表示组合。

语义变化点

没有定义创建组合中部分实例的顺序和方式。

没有定义关联派生和其端的派生之间的逻辑关系。

没有定义关联特化与关联端重定义和子集划分的互相作用。

记法

任何关联都可以画成菱形(比线上的终结符大)加上一条实线，关联的每个端点将此菱形连接到作为端的类型的类目。有多于两个端的关联只能这样画出。

二元关联通常画成一条连接两个类目的实线，或者一条实线将单个类目连接到它本身(这两个端有区别)。一条线可以包含一个或多个连接段。线的单个连接段没有语义意义，但是对于建模工具从图形角度上来说，对关联符号的拖动或者改变大小可以有意义。

关联符号可以图示如下：

a) 关联的名称作为名称串图示在关联符号附近，但不能靠端太近以免与该端的名称混淆。

b) 关联的名称前用斜线标出，或者，当没有画出名称时，就标注派生中的关联。

c) 性质串可以放在关联符号的近旁，但不能靠端太近以免与端处的性质串混淆。

d) 在用实线画出的二元关联中，实心的三角形箭头紧靠或者取代关联的名称，沿直线指向另一个端点，这表示这个端点是有序关联端点中的最后一个。箭头指明此关联应该理解为从远离箭头方向的端到箭头指向的端(见图 74)。

e) 关联之间的泛化可以图示为关联符号之间的泛化箭头。

f) 关联端是在描绘关联的线段和描绘被连接的类目的图标之间的连接。名称串可以放在该线段附近来图示关联端的名称。名称是可选的并可加以抑制。

g) 许多其他记法可以放在线段的端旁。

h) 势域。

i) 以花括号括起的性质串。以下性质串可适用于关联端：

{subsets＜property-name＞}图示该端是称为＜property-name＞-的性质的子集。

{redefines＜end-name＞}图示该端重定义一称为＜end-name＞的端名。

{union}图示该端由作为其子集的联合导出。

{ordered}图示该端表示一有序集。

{bag}图示该端表示允许同一元素可多次出现的汇集。

{sequence}或{seq}图示该端表示一序列(一有序袋)。

当此端可导航时，则任何性质串适用于属性。

注意在缺省情况下，关联端表示一个集合。

关联端上的刺状箭头指明该端是可导航的。关联端处的小 x 指明该端不可导航。在可导航端上，可以加上一个可视性符号作为特征化类目的属性来图示该端的可见性。

如果关联端是派生的，则可在其名称前加一斜线，如果没名称，则用斜线代替。

属性记法可适用于可导航的关联端的名称。

图示组合聚合的记法与二元关联用的记法相同，但聚合端有一个实心菱形。

表示选项

当两条线相交时，交叉可选地以一个小半圆转向绘出，表示线之间不相交(和电路图一样)。

许多选项可以用来在图上绘出导航箭头。实际上，通常习惯抑制一些箭头和交叉，只在以下例外情况下绘出：

a) 显示所有箭头和 x。以显式表明导航的存在与否。

b) 隐藏所有箭头和 x。不绘出任何导航引用。这类似于在视图中抑制信息。

c) 抑制在双向可导航的关联的箭头，只图示具有单向导航性的关联的箭头。这种情况下，双向可导航性和完全没有导航就不能区分了；不过后一种情况实际上很少见。

如果对同一聚集有多个聚合，就可以将聚合的端合并到一单个段而画成一棵树。该点上的任何图示都适用于所有的聚合端。

样式指南

各线段可采用不同样式画出，包括直角段、倾斜段和弯曲段。具体样式由用户自己决定。

关联之间的泛化最好以不同于关联所用的颜色或线宽来绘出。

示例

图 74 所示为一个从 Player 到 year 的称为 PlayedInYear 的二元关联。实心三角形指明阅读顺序：Player PlayedInYear Year。

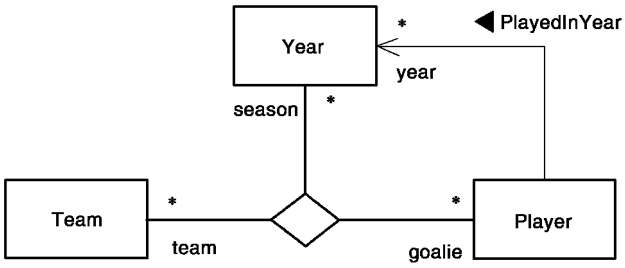


图 74　二元与三元关联

该图更进一步图示出 Team、Year 和 Player 间的三元关联，各端分别以 team、season 和 goalie 命名。

图 75 图示了带有多种修饰的关联端。

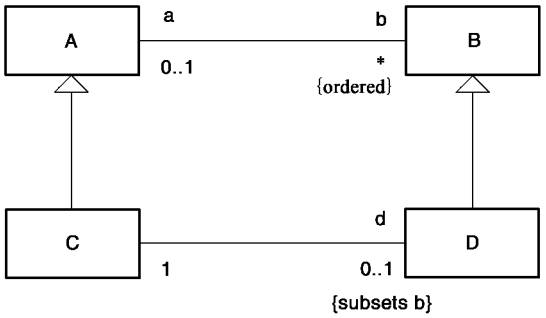


图 75　具有多种标注的关联端

图 75 中的四个关联端的标注解释如下：

a)　名称 a、b 和 d 在三个端上。

b)　势域 0..1 在 a 上，* 在 b 上，1 在未命名的端上，0..1 在 d 上。

c)　在 b 上的定序规约。

d)　在 d 上定义了子集。例如类 C 的实例，汇集 d 是汇集 b 的一个子集。这与 OCL 约束等价。

context C inv：b->includesAll(d)

图 76 用于可导航端的记法。

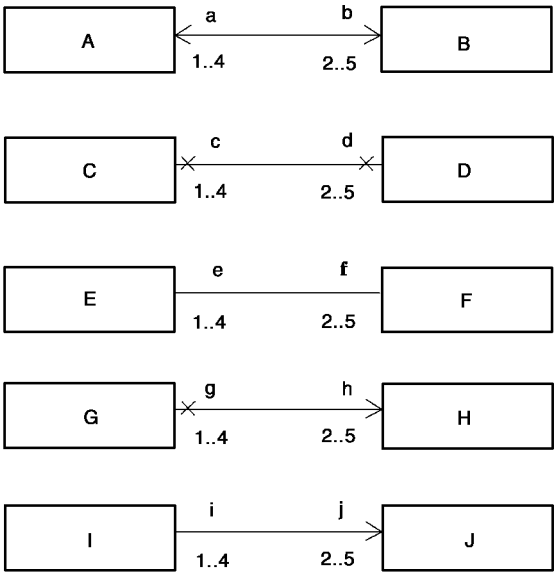


图 76　可导航端的例子

在图 76 中：

a)　顶部的一对 AB 图示带有两个可导航端的二元关联；

b)　第二对 CD 图示带有两个不可导航端的二元关联；

c)　第三对 EF 图示带有两个未规定导航性的二元关联；

d)　第四对 GH 图示一个端可导航而另一端不导航的两个点的二元关联；

e)　第五对 IJ 图示一个端可导航而另一端未规定的二元关联。

图 77 图示采用属性记法的可导航端。一个可导航端是一个属性，因此它可以用属性记法来图示。一般来说，这种记法要和带箭头的线段记法一起使用，这样非常清晰地表明了可导航端也能作为属性。

图 77　用属性记法图示的可导航端的例子

图 78 图示用于派生联合的记法。属性 A::b 由作为子集的所有属性构成的严格联合体派生而来。在这一情况下，只有其中一个属性 A1::b1。所以作为类 A1 的一个实例，b1 是 b 的一个子集，同时 b 由 b1 派生出来。

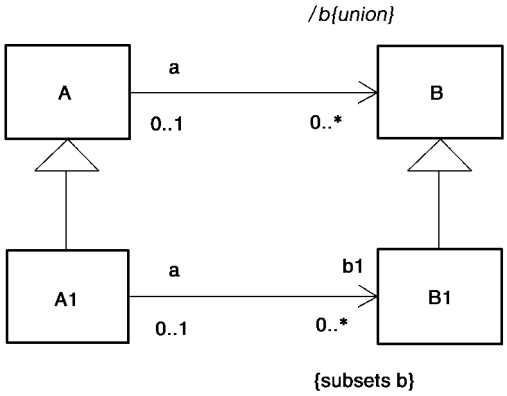


图 78　一个派生联合体的例子

图 79 图示用于组合聚合的黑色菱形记法。

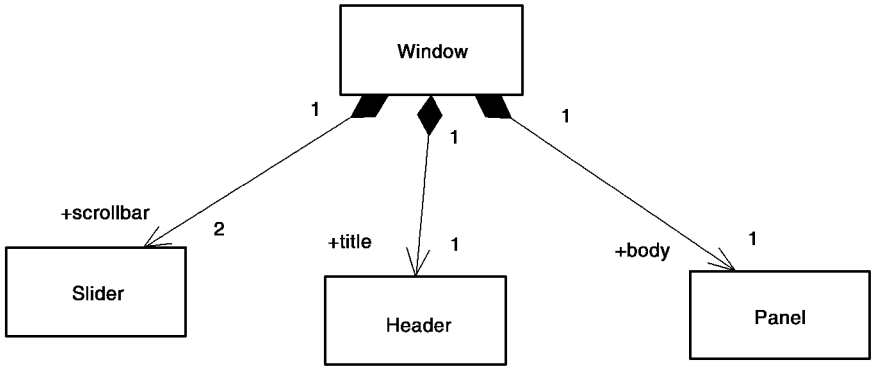


图 79　绘作黑色菱形的复合聚集

9.3.2　类[Class(特化的)]

类描述一个共享相同特征、约束和语义的规约的对象的集合。Constructs::Class 合并了 Constructs::Classifier 和 Basic::Class 的定义。

描述

类是一种其特征是属性和操作的类目。类的属性由该类所拥有性质的实例来表示。有些属性可以表示二元关联的可导航端点。

属性

IsAbstract::Boolean——重定义 Basic::Class 和 Abstractions::Classifier 中相应的属性。

关联

a) ownedAttribute:Property[*]——这些属性(也就是性质)由该类所拥有。这是个有序关联。Classifier::attribute 和 Namespace::ownedMember 的子集。

b) ownedOperation:Operation[*]——由该类所拥有的操作。这是个有序关联。
Classifier::feature 和 Namespace::ownedMember 的子集。

c) spuerClass:Class[*]——它给出了一个类的超类。它重定义 Classifier::general。

约束

无附加的约束。

附加操作

[1] 重载继承操作以排除重定义的性质。

Class::inherit(inhs:Set(NamedElement)):Set(NamedElement);

Inherit=inhs->select(ocllsKindOf(RedefinableElement))->select(redefinedElement->includes(inh)))

语义

类的目的是规定对象的分类,同时规定表征这些对象结构和行为的特征。

类对象应按照该属性(比如其类型和势域)的特性,包括作为这个类的成员的每个属性的值。

当一个对象在一个类中例示时,对具有规定默认值的类的每个属性,如果对该例示该属性的初始值没有显式规定,那么对默认值规约求值以设置该对象的属性的初始值。

只要给定该操作的参数的一组特定代换,类的操作可以对对象调用。操作调用可以导致该对象的属性值的变化。操作的调用也可返回一个作为结果的值,其中结果的类型已经在操作中定义。操作调用也可以直接或间接地引起其他可导航到该操作对其调用的对象的属性的值的变化,导航到从其参数可导航的对象,或导航到操作执行范围中的其他对象。操作调用也可以创建和删除对象。

记法

类是用类目符号来图示的。由于类是最广泛使用的类目,因此不需要在名称前面以双尖号括起的单词"class"。不在双尖号中的元类的类目符号指明一个类。

表示选项

类通常由 3 个格图示。中间格列出属性的表,底格列出操作的表。

属性或操作可以按可见性分组显示出来。表示可见性的关键词或符号能以相同可见性一次给出多个特征。

另外一个格子可以用来图示诸如约束之类的其他细节,或者划分特征。

样式指南

a) 将类名居中加粗;

b) 将类名首字母大写(如果字符集支持大写);

c) 属性和操作以普通字体左对齐;

d) 属性和操作名以小写字母开头;

e) 抽象类类名用斜体;

f) 于必要时图示完整的属性和操作,或者,仅引用一个类将其抑制在其他语境。

示例

如图 80、图 81 所示。

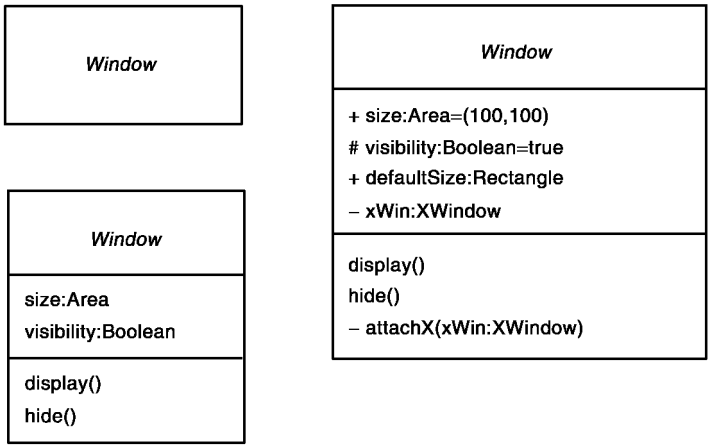


图 80　类记法:抑制的细节、分析层的细节和实现层的细节

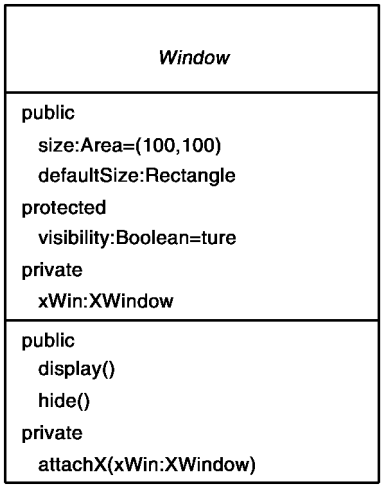


图 81　类记法:按可见性分组的属性和操作

9.3.3　类目(附加性质)

描述

在类目图中定义 Constructs::Classfier。类目是一个类型。类图加进类目和表示类目的属性之间的关联。

属性

无附加属性。

关联

attribute:Property[*]引用作为该类目的(即不是继承也不是引入的)属性的所有的性质。Classifier:feature 的子集,并且是一个派生联合。

约束

无附加的约束。

语义

类目的所有实例都有相应于这个类目的属性的值。

语义变化点

聚集的准确生存周期的语义就是语义变化点。

记法

属性可以用文本串图示，这些串可以句法分析为属性的各种性质。基本语法是(可选部分以括号括起)：

[*visibility*][/]*name*[:*type*][*multiplicity*][=][{*property-string*}]

[可见性][/]名称[:类型][势域][=默认值][{性质串}]

下面分项描述每个部分：

a) *visibility* 是可见性符号，如+or−。见“可见性种类”。

b) /指明该属性是派生的。

c) *name* 是该属性的名称。

d) *type* 标识作为该属性的类型的类目。

e) *multiplicity* 图示方括号中的属性的势域。当势域为1时，这个术语可以省略。见“势域元素”。

f) *default* 是该属性的一个或多个默认值的一个表达式。

g) {*property-string*}指明适用于该属性的性质值。性质串是可选项(不规定任何性质时省略掉括号)。

下面的属性串可适用于如下属性：{readOnly}，{union}，{subsets＜property-name＞}，{redefines＜property-name＞}，{ordered}，{bag}，{seq}或{sequence}，和{composite}。

与已经继承的属性具有相同名称的属性解释为重定义，而无需性质串{redefines＜x＞}。注意，重定义的属性在重定义时并没有继承进命名空间，所以它的名称可以在特征化的类目中重用，或用于重定义中的属性，或用于某些其他属性。

表示选项

类型、可见性、默认值、势域和性质串都可加以抑制而不被显示，即使该模型中有值。

属性的单个性质能在列中图示，而不是作为连续的串。

样式指南

属性名一般以小写字母开头。多词名称通常将各单词拼接形成，除第一个单词外，每个单词的首字母大写，其余字母均小写。

示例

如图82所示。

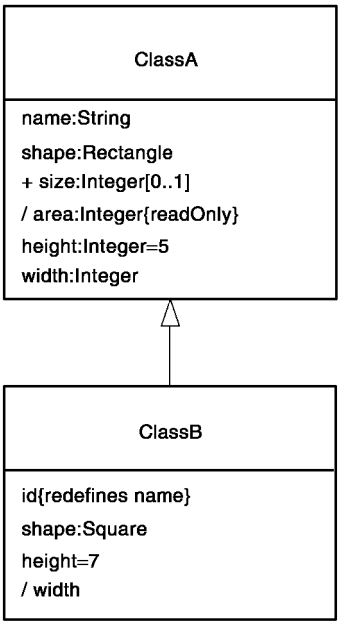


图 82　属性的例子

对图 82 中的属性解释如下：

a) ClassA::name 是一个 String 类型的属性。

b) ClassA::shape 是一个 Rectangle 类型的属性。

c) ClassA::size 是带势域 0..1 的 Integer 类型的公有属性。

d) ClassA::area 是带有 Integer 类型的派生属性。标注为只读。

e) ClassA::height 是 Integer 类型的属性，默认值为 5。

f) ClassA::width 是 Integer 类型的属性。

g) ClassB::id 是重定义 ClassA::name 的属性。

h) ClassB::shape 是重定义 ClassA::shape 的属性。它是 Square 类型，是 Rectangle 的特化。

i) ClassB::height 是重定义 ClassA::height 的属性。它的默认值对实例 ClassB 为 7，此实例覆盖 ClassA 的默认值 5。

j) ClassB::width 是重定义 ClassA::width 的一个派生属性，ClassA::width 不是派生的。

属性也可以用关联记法来图示，如图 83 所示，箭头尾部没有装饰。

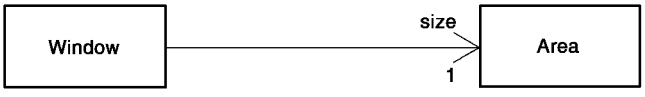


图 83　对属性的类似关联记法的图示

9.3.4　操作[Operation(附加性质)]

描述

在操作图中定义了 Constructs::Operation。类图为操作和表示拥有该操作的类加入了关联。

属性

无附加属性。

关联

class:Class[0..1]重定义了在基本包中的相应的关联。RedefinableElement::redefinitionContext、NamedElement::namespace 和 Feature::featuringClassifier 的子集。

约束

无附加的约束。

语义

一个操作可以为一个类的命名空间拥有也可以包含在其中,这个类的命名空间为操作重定义提供了语境环境。

9.3.5 性质[Property(特化的)]

性质是类目的结构特征,它表征该类目的实例。Constructs::Property 合并了 Basic::Property 和 Constructs::StructuralFeature 的定义。

当性质为类所拥有时表示属性。在这种情况下,它使类的一个实例与该属性的类型的一个或一组值有关。

当性质为关联所拥有时,表示该联系的不可导航的端。在这种情况下性质的类型是关联的端的类型。

描述

性质借助于与一个或多个值的命名联系了表示一个或多个实例声明的状态。当性质是类目的一个属性时,通过将实例保有在槽内使一个或多个值和类目实例发生联系。当性质是一个关联端时,值和此关联的另一端的实例相关(见"关联的语义")。

性质是 Constructs::TypedElement 的一个间接子类。性质所表示的有效值的范围可以通过设置该性质的类型来控制。

属性

a) idDerivedUnion:Boolean——规定该性质是否是派生为所有性质的联合,这些性质被约束为其子集。默认值是 false。

b) IsReadOnly:Boolean——重定义了 Basic::Property 和 Abstractions::StructuralFeature 中相应的属性。默认值是 false。

关联

a) association:Association[0..1]——引用该性质为其成员的关联(如果有时);

b) owningAssociation:Association[0..1]——引用该性质拥有的关联(如果有时);Property::association,NamedElement::namespace 和 Feature::featuringClassifier 的子集;

c) RedefinedProperty: Property[*]——引用该性质重定义的性质;RedefinableElement::redefinedElement 的子集;

d) SubsettedProperty:Property[*]——引用该性质约束为子集的性质;

e) /opposite:Property[0..1]——当性质是一个两端都可导航的二元关联的一个可导航端时,给出了另一端。

约束

[1] 如果这个性质由一个类所拥有,与一个二元关联相关联,且该关联的另一端也由一个类拥有,那么 opposite 给出另一端。

Opposite=

if owningAssociation->notEmpty() and association.memberEnd->size()=2 then

```
  let otherEnd=(association.memberEnd-self)->any() in
    if otherEnd.owningAssociation->notEmpty() then otherEnd else Set {} endif
else Set {}
endif
```

[2] 一个组合聚合的势域应没有大于1的上界。

```
IsComposite implies(upperBound()->isEmpty() or upperBound()<=1)
```

[3] 划分子集仅可出现在被划分子集的性质和划分的性质的语境一致的时候。

```
SubsettedProperty->notEmpty() implies
  (subsettingContext()->notEmpty() and subsettingContext()->forAll(sc|
    subsettedLProperty->forAll(sp|
      sp.subsettingContext()->exists(c|sc.conforms To (c)))))
```

[4] 可导航性质(由类拥有的)只能由可导航性质重定义或细划为子集。

```
(subsettedProperty->exists(sp|sp.class->notEmpty())
      implies class->notEmpty())
and
(redefinedProperty->exists(rp|rp.class->notEmpty())
      implies class->notEmpty())
```

[5] 一个划分子集的性质可以加强被划分为子集的性质的类型,而且其上界可以更小。

```
SubsettedProperty->forAll(sp|
type.conformsTo(sp.type) and
  ((upperBound()->notEmpty() and sp.upperBound()->notEmpty()) implies
    upperBound()<=sp.upperBound()))
```

[6] 只有可导航性质可以标注为只读。

```
IsReadOnly implies class->notEmpty()
```

[7] 一个派生联合是派生的。

```
isDerivedUnion implies isDerived
```

[8] 一个派生联合是只读的。

```
IsDerivedUnion implies isReadOnly
```

附加操作

[1] 查询isConsistentWith()为在语境中可以重定义的任意两个性质规定其重定义是否在逻辑上一致。如果被重定义的性质和重定义的性质的类型一致,重定义的性质的势域(如果规定)包含在被重定义的性质的势域里,且当重定义的性质是派生的时被重定义的性质也是派生的,那么被重定义的性质和重定义的性质就是一致的。

```
Property::isConsistentWith(redefinee:RedefinableElement):Boolean
Pre:redefinee.isRedefinitionContextValid(self)
IsConsistentWith=(redefinee.oclIsKindOf(Property) and
let prop:Property=redefinee.oclAsType(Property) in
  type.conformsTo(prop.type) and
  (lowerBound()->notEmpty and prop.lowerBound()->notEmpty() implies
(lowerBound()>=prop.lowerBound()) and
  (upperBound()->notEmpty and prop.upperBound()->notEmpty() implies
    upperBound()<=prop.upperBound()) and
  (prop.isDerived implies isDerived)
)
```

[2] 查询 subsettingContext()给出了将性质划分为子集的语境。当作为属性时,它由相应的类目组成;作为关联的端时,它由另一端的所有类目组成。

Property::subsettingContext():Set(Type)

SubsettingContext=

If association->notEmpty()

Then association and Type-type

Else if classifier->notEmpty() then Set{classifier} else Set{} endif

Endif

语义

当性质是通过拥有的属性被类或被数据类型拥有时,表示该类或数据类型的一个属性。当通过所拥有的端为一个关联拥有时,表示该关联的一个不可导航的端。当在上述任何一种情况下例示时,性质表示某个与一个(或者,在三次或更高次关联中,多于一个)类型关联的值或值的汇集。这些类型集合称为该性质的语境。当作为属性时,语境是拥有的类目;当作为关联的端时,语境是该关联的另一端或两端的类型的集合。

对语境实例的性质例示的值或值的汇集与该性质的类型一致。性质从势域元素继承,所以允许规定势域的界限。这些界限约束了汇集的大小。典型的默认的最大界是 1。如表 1 所示。

性质也继承了 isUnique 和 isOrdered 元属性。当 isUnique 为 ture(缺省)时,值的汇集可不包含重复。当 isOrdered 为 true(缺省是 false)时,值的汇集有序。综合这两种情况,使性质的类型能以表 1 方式表示汇集。

表 1 性质的汇集类型

isOrdered	isUnique	Collection type
false	true	Set
true	true	OrderedSet
false	false	Bag
true	false	Sequence

如果对一个性质规定有默认值,则当该性质的一个特定的设置不存在,或者该模型中要求该性质有一个特定的值的情况下创建该性质的一个实例时,对该默认值求值。求出的默认值即作为该性质的初始值。

当导出一个性质时,其值可从其他信息计算出来。涉及派生的性质的动作,与非派生性质的行为相同。派生出的性质通常规定为只读的(就是说,客户不能直接改变其值)。但当派生的性质的值可改变时,需要一种实现来适当地改变导出的源信息。对派生的性质的导出可由约束来规定。

一个性质的名称和可见性不需要与由其重新定义的性质的名称和可视性相匹配。

导出的性质可以重新定义一个未被导出的性质。应有一种实现来保证:在更新该属性时,应保持由此派生所隐含的约束。

如果一个性质有规定的默认值,并且该性质重新定义带有规定默认值的另一性质,则这个重定义性质用于将被定义的默认值替代为更通用的默认值。

如果一个可导航的性质(属性)标注为“只读”,则它赋予一个初始值后即不能被更新。

一个性质可标注为另一性质的子集,只要划分性质的语境中的每一元素符合于被划分子集的性质的语境的对应元素。在这种情况下,和划分子集的属性的实例关联的汇集应包含在与被划分子集对应实例相关联的汇集之内(或两者相等)。

一个性质可标注为导出的联合。这意味着,在某些语境下,由该性质指代的值的汇集,是由在相同语境下,对前划分子集的性质所指代的全部值的严格的联合所导出的。如果该性质具有的势域上界为 1,则意味着所有它的子集的值应为空或相同。

记法

性质的记法对其用作属性和关联端独自定义。划分子集和派生的联合的例子对关联图示出来。

9.3.6 类目图(Classifiers diagram)

构造包的类目图规定以下概念:类目、类型化元素、势域元素、可重定义元素、特征和结构特征。在每种情况下,这些概念都可以从其对应的定义加以扩展和重定义。如图 84 所示。

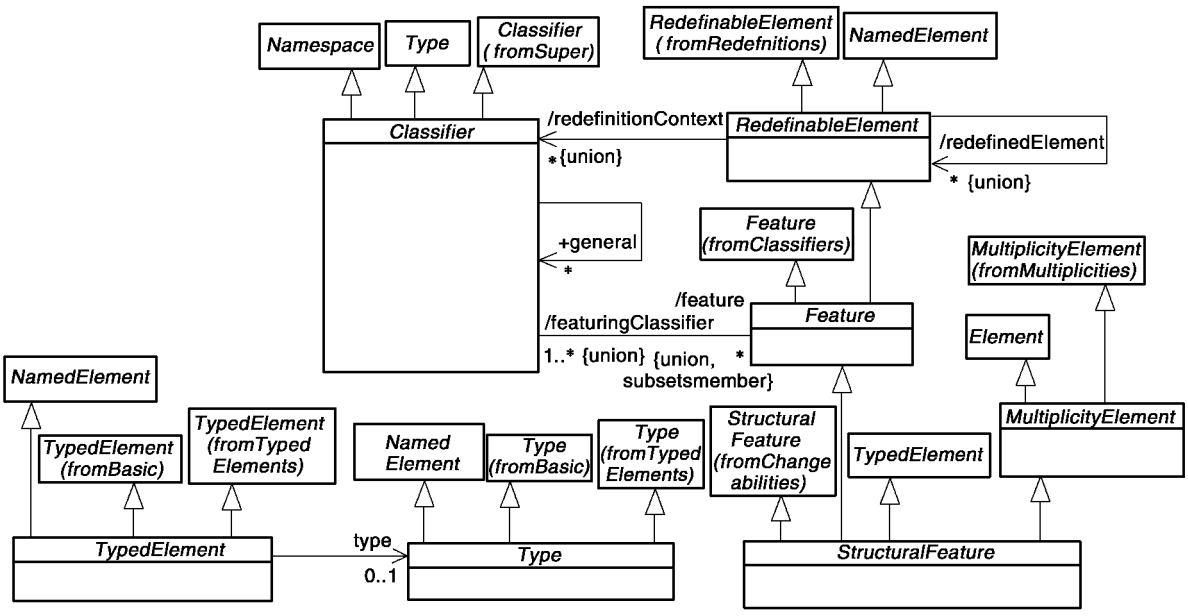


图 84 构造包的类目图

9.3.7 类目[Classifier(特化的)]

描述

Constructs::Classifier——合并了源自基本层和抽象层的类目的定义。它添加来自 Constructs::Namespace 和 Constructs::Type 的特化。

属性

无附加属性。

关联

feature:Feature[*]——重定义了在抽象层中相应的关联。子集 Namespace::member 是一个派生的联合。注意,可以有类目的成员,它们具有该类型特征,但却并没有包含在这一关联之中,比如,继承来的特征。

约束

无附加约束。

语义

无附加语义。

记法

同抽象层中所定义的。

9.3.8 特征[Feature(特化的)]

描述

Constructs::Feature——重用了源自抽象层的特征的定义,并且加上来自 Constructs::RedefinableElement 的特化。

属性

无附加属性。

关联

featureClassifier:Classifier[1..*]——重定义了抽象层中相应的关联。这是一个派生联合。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.3.9 势域元素[MultiplicityElement(特化的)]

描述

Constructs::MultiplicityElement——重用了源自抽象层的势域元素的定义,并且加上了源自 Constructs::Element 的特化。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.3.10 可重定义的元素[RedefieableElement(特化的)]

描述

Constructs::RedefieableElemen——重用了在抽象层中对可重定义元素的定义,并且添加了来自 Constructs::NamedElement 的特化。

属性

无附加属性。

关联

a) /redefinedElement:RedefinableElement[*]——该派生联合从抽象层中重定义而来;

b) /redefinitionContext:Classifier[*]——该派生联合从抽象层中重定义而来。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.3.11 结构特征[StructuralFeature(特化的)]

描述

Constructs::StructuralFeature——重用了源自抽象层的结构特征的定义,并且添加了源自Constructs::Feature,Constructs::TypedElement,Constructs::MultiplicityElement的特化。

通过将势域元素特化,它为与结构特征的例示相关联的值的集合的有效的势的势域提供支持。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.3.12 类型[Type(特化的)]

描述

Constructs::Type——合并了源自基础层和抽象层中类型的定义,并且添加了源自Constructs::NamedElement的特化。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.3.13 类型元素[TypedElement(特化的)]

描述

Constructs::TypedElement——合并了源自基础层和抽象层中类型元素的定义,并且添加了源自Constructs::NamedElement的特化。

属性

type:Classifier[1]——重定义了基础层和抽象层中相对应的属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

与抽象层中定义的记法相同。

9.4 约束图(Constraints diagram)

构造包的约束图规定了约束构造,并且为命名空间构造添加了特征,如图 85 所示。

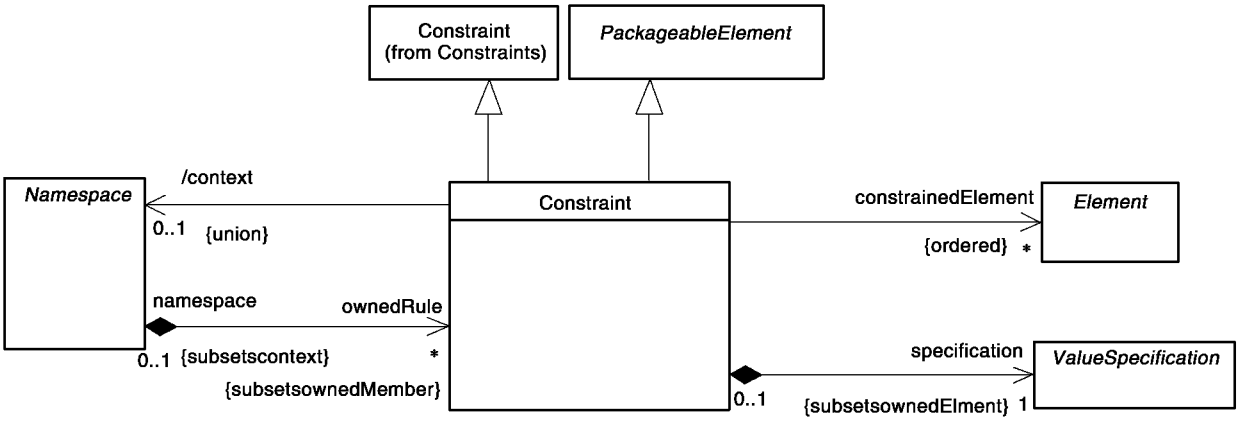


图 85 构造包的类图

9.4.1 约束(Constraint)

描述

Constructs::Constraint——重用了源自 Abstractions::Constraints 的约束定义,并且为 PackageableElement 添加了特定的说明。

属性

没有附件属性。

关联

a) constrainedElement:Element 重定义了抽象层中的相应性质;

b) context:Namespace[0..1]重定义了抽象层中的相应性质,它是一个导出联合;

c) Specification: ValueSpecification 重定义了抽象层中的相应性质,并且对 Element.ownedElement 作了子集划分。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.4.2 命名空间[Namespace(附加性质)]

描述

Constructs::Namespace 在命名空间图中定义。约束图图示了命名空间和由命名空间表示的约束之间的关联。

属性

无附加属性。

关联

ownedRule:Constraint[*]重定义了抽象层中相应的性质,并且对 Namespace::ownedMember 做了子集划分。

约束

无附加约束。

语义

无附加语义。

9.5 数据类型图(DataTypes diagram)

构造包的数据类型图规定数据类型、枚举、枚举文字及原子类型构造,并且为性质和操作构造添加了特征。这些构造用来定义原子数据类型(比如整数和串),及用户定义的枚举数据类型。这些数据类型主要用来声明类的属性的类型。如图 86 所示。

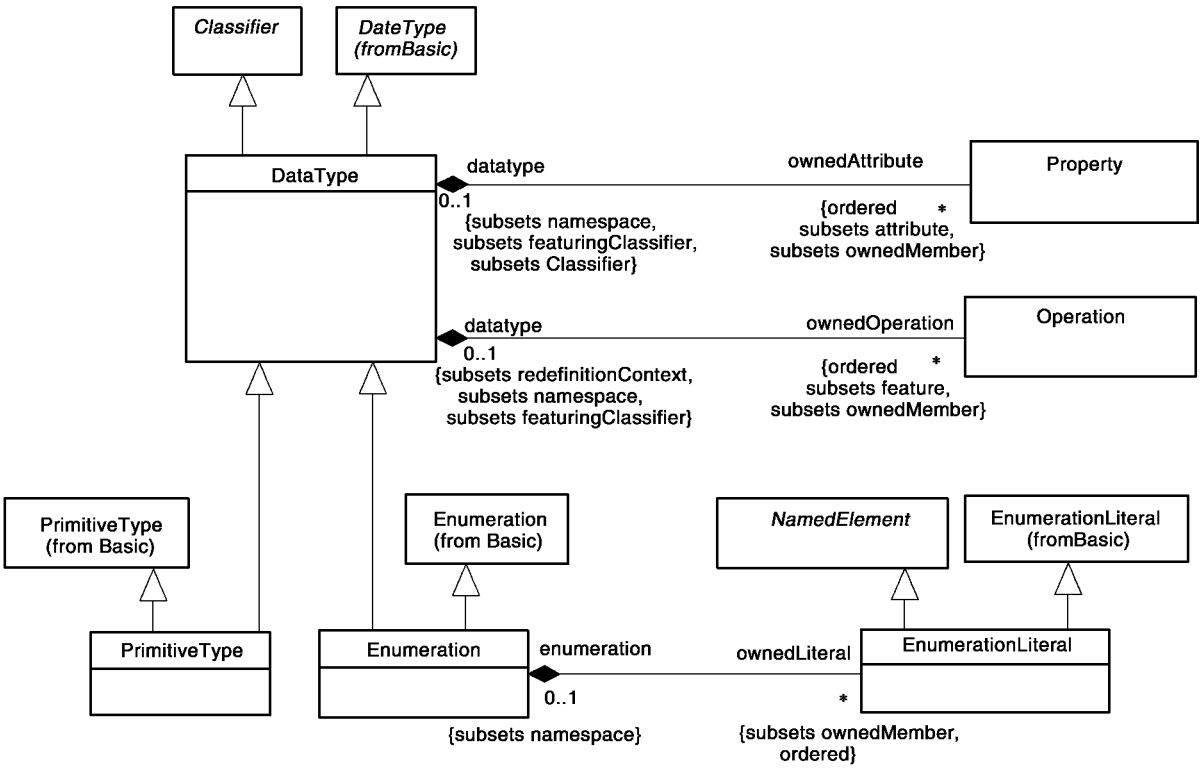


图 86 数据类型图中定义的类

9.5.1 数据类型[DataType(特化的)]

数据类型是一种其值没有标识的类型(即是纯粹值)。数据类型包括原子的内置类型(比如整数和串)及枚举类型。

描述

Constructs::DataType——重用了源自基础层中数据类型的定义,并且为 Constructs::Classifier 添加了特化。

数据类型定义了一种类目,其中的操作都是纯函数(即可以返回数据值但不能改变数据值,因为它们都没有标识)。例如,对一个数与作为变元的另一个数的"加"操作产生作为结果的第 3 个数,目标和变元都没有改变。

数据类型也可以包含一些属性,以支持对结构化的数据类型进行建模。

属性

无附加属性。

关联

a) ownedAttribute:Attribute[*]——由数据类型所拥有的属性。对 Classifier::attribute 和 Element::ownedMember 进行了子集划分。

b) ownedOperation:Operation[*]——由数据类型所拥有的操作。对 Classifier::feature 和 Element:ownedMember 进行了子集划分。

约束

无附加约束。

语义

数据类型是一种特别种类的类目,和类相似,其实例是值而不是对象。例如,整数和串通常作为值来处理。值没有标识,所以,同一值出现两次时无法做出区分。通常,数据类型用作一个属性的类型规约。枚举类型是一个由有限个值组成的用户定义的类型。

如果一个数据类型具有属性,那么,该数据类型的实例就包含与该属性匹配的属性值。

语义变化点

对数据类型能力的任何限制,比如对其属性的类型加以约束,都是一个语义变化点。

记法

数据类型由具有关键字《data Type》的矩形符号来指代,或者,当它由一个比如属性来引用时,由一个包含该数据类型的名称的串来指代。

表示选项

经常对属性格子进行抑制,特别当一个数据类型不包含属性时。操作格子也可以加以抑制。对失去的格子不会绘制一条分隔线。如果一个格子被抑制,就不能画出对其中元素的出现与否的引用。必要时,可以用格子名称来避免歧义。

可附加格子来图示预定义的或用户定义的性质(比如图示商业规则、责任、变化、处理的事件、引起的异常)。大多数格子只不过是串的列表,当然,也可用更为复杂的格式。每一个格子的出现都隐式地基于其内容。在需要时可以使用格子名。

带有衍型图标的数据类型符号可予以"套缩",仅图示为衍型图标,数据类型的名称或者在矩形内或者在图标下。抑制数据类型的其他内容。

样式指南

用粗体字将输入数据类型的名称居中放置。

以双尖括号将普通字体的关键字(包括衍型的名称)括起,在数据类型名上面居中放置。

当使用区分大小写的语言时,将名称大写(即以大写字母开始)。

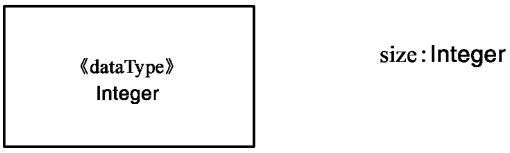
以普通字体书写属性和操作并左对齐。

属性和操作的名称应该以小写字母开头。

在需要时图示所有的属性和操作，在其他语境或引用中予以抑制。

示例

如图 87 所示。



（左边是指代数据类型的图标，右面是对用于属性的数据类型的一个引用）

图 87　数据类型记法

9.5.2　枚举[Enumeration(特化的)]

枚举是一种其值在模型中作为枚举文字加以枚举的数据类型。

描述

Constructs::Enumeration——重用了源自基础层中的枚举类型的定义，并且对 Constructs::DataType添加了特化。

枚举时一种数据类型，其实例可以是预定义的任意数目的枚举文字。

可以将可应用的枚举文字的集合扩展到其他包或外廓。

属性

无附加属性。

关联

ownedLiteral:EmumerationLiteral[*]——该枚举的文字的有序集合，并且对 Element::ownedMember 进行子集划分。

约束

无附加约束。

语义

枚举的运行时实例是数据值。每一个这样的值都与恰好一个枚举文字相对应。

记法

枚举可使用带有关键字《enumeration》的类目记法(矩形)来图示。枚举的名称放在名称格子的上面。该枚举属性的列表放在名称格子的下面。该枚举的操作的列表放在属性格子的下面。该枚举文字的列表可一次一行放在底部格子。属性和操作的格子可加以抑制，如果它们为空，则一般加以抑制。

示例

如图 88 所示。

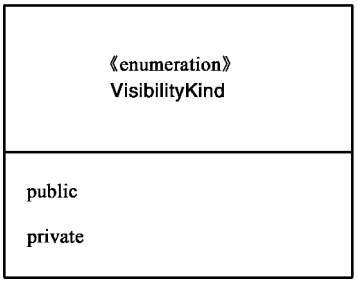


图 88　一个枚举的例子

9.5.3　枚举文字[EnumerationLiteral(特化的)]

枚举文字是用户为某个枚举定义的数据值。

描述

Constructs::EnumerationLiteral——重用了源自基础层的枚举的定义，并且为 Constructs::NamedElement 添加了特化。

属性

无附加属性。

关联

enumeration:Enumeration[0..1]——这一枚举文字是其一个成员的枚举，是 NamedElement::namespace 的子集。

约束

无附加约束。

语义

EnumerationLiteral——定义枚举数据类型的运行时扩展的一个元素。

EnumerationLiteral——有一个在其枚举数据类型中用来对其标识的名称。这个名称在它属于的枚举中惟一，但却并不是全局的，且应对一般使用加上限定。

与枚举文字可以对等式进行比较。

记法

EnumerationLiteral——一般一次一行图示为该枚举记法的格子内。见“枚举(特化的)”。

示例

见“枚举(特化的)”。

9.5.4　操作[Operation(附加性质)]

描述

Constructs::Operation 在操作图中定义。数据类型图图示了操作和数据类型之间的一种关联，这种关联由数据类型表示对某个操作的所属关系。

属性

无附加属性。

关联

datatype:DataType[0..1]——拥有这一操作的数据类型，并且对 NamedElement::namespace 和 Feature::featuringClassifier 以及 RedefinableElement::redefinitionContext 进行了子集划分。

约束

无附加约束。

语义

操作可以属于某个数据类型的命名空间并为其所拥有，该数据类型为可能的重定义提供语境。

9.5.5 原子类型[PrimitiveType(特化的)]

原子类型定义了一个没有任何相关子结构(即没有部件)的预定义数据类型。原子类型可以有定义在 UML 之外的代数和操作，例如以数学方式。

描述

Constructs::PrimitiveType 重用了源自基础层中原子类型的定义，并且为 Constructs::DataType 添加了特化。

在 UML 中所用的原子类型的实例包括：布尔值、整型、不受限自然数和串(见第 10 章“核心::原子类型”)。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

原子类型的运行时实例是数据值。这些值和在 UML 之外定义的数学元素(例如，各种整数)存在多对一的对应。

原子类型的实例没有标识。如果两个实例具有相同的表示，则不能做出区分。

记法

在原子类型的名称上面或前面，有关键字《Primitive》来标识。

预定义的原子类型的实例(见第 10 章“核心::原子类型”)可以使用与用于对这一实例的引用相同的记法来指代(见“值约束”的子类型)。

示例

见第 10 章“核心::原子类型”的例子。

9.5.6 性质[Property(附加性质)]

描述

Constructs::Property 在类图中定义。数据类型图图示性质和数据类型之间的一种关联，这种关联由数据类型表示对性质的所属关系。

属性

无附加属性。

关联

datatype:DataType[0..1]——拥有该属性的数据类型，并且对 NamedElement::namespace，Feature::featuringClassifier 和 Property::classifier 进行子集划分。

约束

无附加约束。

语义

性质可以存在于数据类型的命名空间之内并为其拥有。

9.6 命名空间图(Namespaces diagram)

构造包的命名空间图规定命名空间及有关构造。它规定命名的元素如何定义为命名空间的一个成员,同时也规定引入所有或者单个包成员的任一命名空间的能力。如图 89 所示。

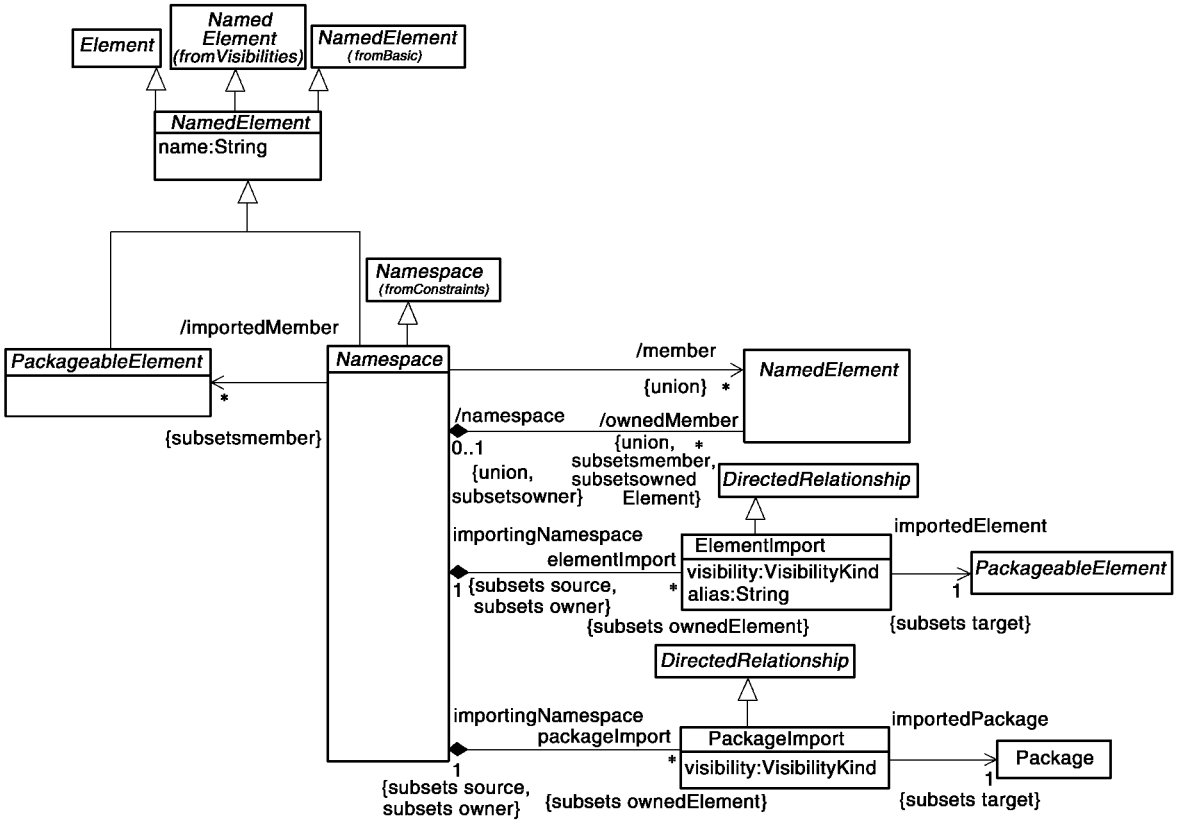


图 89 构造包的命名空间图

9.6.1 元素引入(ElementImport)

元素引入标识另一包中的一个元素,并且允许使用其名称而不使用限定符来引用该元素。

描述

元素引入定义为引入命名空间和可打包元素之间的一种有向的关系。可打包元素的名称或其别名添加到引入命名空间。它也控制被引入的元素能否允许进一步被引用。

属性

a) visibility:VisibilityKind——规定被引入的可打包元素在引入包中的可见性。默认的可见性和被引入的元素的可见性相同。如果被引入的元素不具有可见性,则可以对该元素引入添加可见性。

b) Alias:String[0..1]——规定应该添加到引入的包的命名空间中的名称,以代替被引入的可打包元素的名称。所起的别名与这个引入包中任何其他成员的名称不能发生冲突。默认情况下,不使用别名。

关联

a) importedElement:PackageableElement[1]——规定其名称应添加到命名空间的可打包元素,是 DirectedRelationship::target 的子类;

b) ImportingNamespace:Namespace[1]——规定了从另一包引入可打包元素的命名空间,是DirectedRelationship::source 和 Element::owner 的子集。

约束

[1] 元素引入的可见性或者是公有的,或者是私有的。

Self. visibility=#public or self. visibility=#private

[2] 引入元素或者有公有可见性或者没有任何可见性。

Self. importedElement. visibiliby. notEmpty()implies self. importedElement. visibility=#public

附加操作

[1] 查询 getName()返回一个名称,通过这个名称在引入命名空间中找到被引入的可打包元素。

```
ElementImport::getName():String
GetName=
  If self. alias->notEmpty() then
    Self. alias
  Else
    Selft. importedElement. name
  Endif
```

语义

元素引入将一个可打包元素的名称从一个包中添加到另引入的命名空间中。它是通过引用工作,这就意味着不可能将特征添加到元素引入自身,但可以在引入单个元素的命名空间中修改被引用的元素。一个元素引入选择地引入一些独立的元素,而不必依赖于包引入。

当与引入命名空间中的外部名称(一个元素在一个包含的命名空间中定义的元素采用在被包含的命名空间中的非限定名可用)引起名称冲突时,这个外部名称由引入元素隐藏,且非限定名引用被引入的元素。而外部名称可由其限定名来访问。

如果具有相同名称的多个元素作为元素引入或包引入的序列被引入到一个命名空间,则为了便于以后使用这些元素,应该对被引入元素的名称加以限定,并且这些元素不添加到引入命名空间。如果被引入元素的名称和该引入的命名空间所拥有的元素的名称相同,那么这个被引入的元素的名称就应加以限定以便使用,并且该元素不添加到该命名空间。

被引入的元素可以使用元素或者成员引入而被另一命名空间再次引入。

元素引入的可见性可以和被引入的元素的可见性受到相同的或更多的限制。

记法

元素引入使用一个从引入命名空间到被引入元素的虚线加开式箭头来图示。如果可见性是公用的,那么将关键字《import》放在带虚线箭头旁边,否则就使用关键字《access》。

如果元素引入有别名,那么就代替被引入元素的名称使用。别名可以图示在关键字《import》后面或下面。

表示选项

如果被引入的元素是一个包,那么关键字可以可选地放在元素即《elementimports》的后面。

作为对虚线箭头的一种替代,可以通过一个文本来图示一个元素引入,这个文本在命名空间名称的下面或后面的花括号中惟一地标识被引入的元素。这些文本句法如下:

{element import<qualifiedName>}or{element access<qualifiedName>}

别名也可选的图示:

{element import<qualifiedName>as<alias>}or{element access<qualifiedName>as<alias>}

示例

在图 90 中所示的元素引入,允许元素可以不需经过任何限制条件,就能在 Program 包中的元素引

用在没有限定的 Type 中的 Time 类型。然而，这些元素仍需显式引用 Types::Integer，因为这个元素没有被引入。

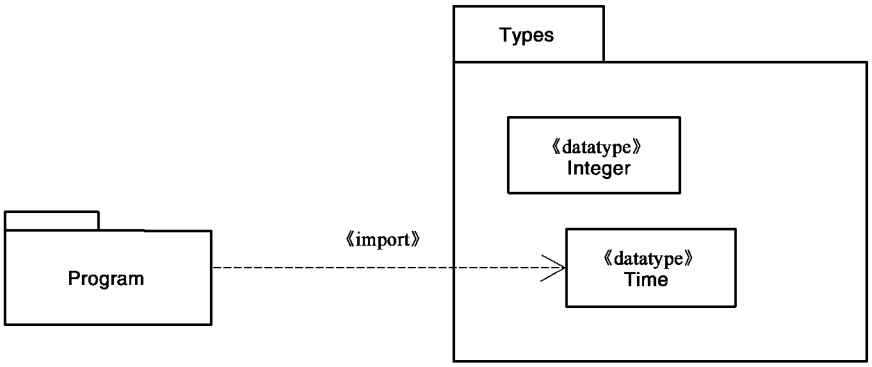


图 90　元素引入的例子

在图 91 中，元素引入和别名复合在一起，这就是说，类型 Types::Real 实数在包 Shapes 中作为双浮点数 Double 来引用。

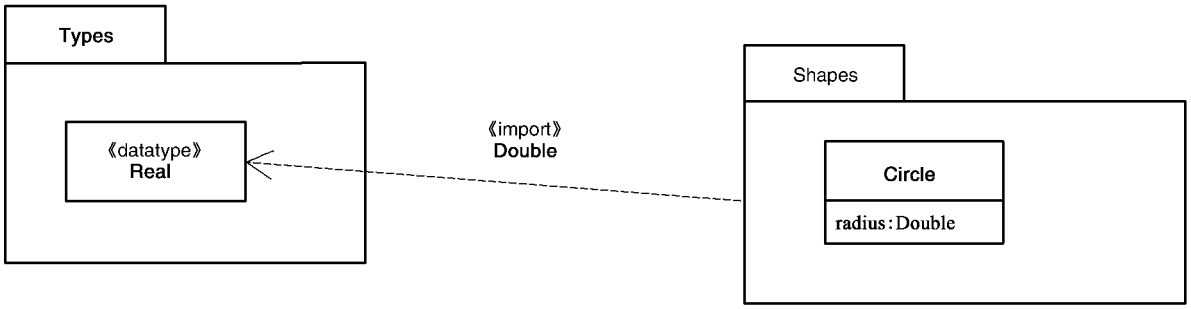


图 91　带别名的元素引入的例子

9.6.2　命名元素[NamedElement(特化的)]

描述

Constructs::NamedElement——重用源自 Abstraction::Visibilities 中命名元素的定义，并且出自 Construcst::Element 和 Basic::NamedElement 添加了特化。

属性

name:String[0..1]——重定义源自 Basic::NamedElement 和 Abstraction::Visibilities::Named-Element 中相应的属性。

关联

namespace:NamedElement[0..1]——拥有该命名元素的命名空间。重定义了源自 Abstraction::Namedspace::NamedElement 的相应性质。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.6.3 命名空间[Namespace(特化的)]

描述

Constructs::Namespace——重用 Abstraction::Constraints::Namespace 的定义。

命名空间有能力引入一个包中的单个成员或所有成员,因此可以不通过在引入的命名空间的限定名来引用那些已命名包中的元素。在出现冲突的情况下,应使用一些被限定名或别名来消除被引用元素具有的歧义。

属性

无附加属性。

关联

a) elementImport: ElementImport[*]——引用由命名空间拥有的元素,是 Element::ownedElement 的子集;

b) ImportedMember:PackageableElement[*]——引用可打包元素,此打包元素是该命名空间的成员,这个包是包引入或元素引入的结果,是 Namespace::member 的子集;

c) member: NamedElement[*]——重定义 Abstraction::Namespaces::Namespace 的相应性质;

d) OwnedMember:NamedElement[*]——重定义 Abstraction::Namespaces::Namespace 中的相应性质;

e) PackageImport: PackageImport[*]——引用该命名空间所拥有的包引入,是 Element::ownedElement 的子集。

约束

[1] 被引入成员的性质是从元素引入和包引入中导出的。

Self. importedMember->includesAll(self. importedMembers(self. elementimport. importedElement. asset()->union(self. p0ackageImport. importedPackage->collect(p|p. visibleMembers()))))).

附加操作

[1] 查询 getNamesOfMember()由于考虑到引入而被重载。它反应了在引入的包中存在的元素的名称集合,要么该方法属于这个元素,如果不属于这个元素的话,就再单独引入它。如果不能单独引入它,就通过某个包来引入。

```
Namespace::getNamesOfMember(element:NamedElement):Set(String);
GetNamesOfMember=
If self. ownedMember->includes(element)
    Then Set{}->include(element. name)
Else let elementImports:ElementImport=self. elementImport->select(ei|ei. importedElement=element)in
    If elementImports->notEmpty()
      Then elementImports->collect(el|el. getName())
    Else
      Self. packageImport->select(pi|pi. importedPackage. visibleMembers()->
      includes(element))->collect(pi|pi. importedPackage. getNamesOfMember(element))
    endif
endif
```

[2] 查询操作 importMembers()定义了某个具体的可打包元素集合被引入到命名空间,但不包括那些隐藏的可打包元素集合,那些集合的名称和现在某些成员的名称相冲突,也不包括那些在引入时有相同名称的集合。

Namespace::importMembers(imps:Set(PackageableElement)):Set(PackageElement);

ImportMembers = self. excludeCollisions (imps)-> select (impl | self. ownedMember-> forAll (mem|

Mem. imp. isDistingguishableFrom(mem,self)L))

[3] 查询操作 excludeCollisions()将同一命名空间中不容易相互区分开的可打包元素区分出来。

Namespace::excludeCollisions(imps:Set(PackageableElements)):SetPackageableElements);

ExcludeCollisions = imps-> reject (imp1 | imps. exists (imp2 | notimp1. isDistinguishableFrom (imp2, self)))

语义

无附加语义。

记法

无附加记法。

9.6.4 可打包元素(PackageableElement)

一个可打包元素表明了一个直接属于某个包的命名元素。

描述

一个可打包元素指明了一个直接属于某个包的元素。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

9.6.5 包引入(PachageImport)

包引入是表示了一种关系,这种关系允许通过一个非限定的名称去引用来自于另一个命名空间的包的成员。

描述

包引入被定义为一种有向的关系,这种关系标识了一个包,这个包的成员被另一个命名空间引用了。

属性

visibility:VisibilityKind 指明了在引入的命名空间中,某个被引入的可打包元素的可见性,例如,引入的元素对那些将可打包元素作为引入包来使用的包是否可见等。如果包引入具有公共属性,那么引入的元素在包外面就具有可见性,反之亦然。在默认情况下,可见性属性的值是公共的。

关联

a) ImportedPackage:Package[1]——规定了某个包,该包的成员被引入了某命名空间中,是 DirectedRelationship::target 的子类;

b) ImportingNamespace:Namespace[1]——规定了某个命名空间,该命名空间从一个包中引入了成员,是 DirectedRelationship::source 和 Element::owner 的子类。

约束

[1] 包引入的可见性要么是公共的,要么是私有的。

Self. visibility=#public or self. visibility=#private

语义

包引入表示了引入的命名空间和一个包之间的关系,表明该引入的命名空间将这个包的成员的名称添加到它自己的命名空间之中。从概念上讲,包引入就是将被引入包中的所有独立元素引入,除非包中的元素已经存在单独的元素引入定义。

记法

一般通过使用从需要引入的命名空间到被引用的命名空间之间的箭头来表示一个成员引入。关键字《import》放在箭头旁边。

包引入通过使用从需要引入的包到被引入的包之间的箭头来表示。相应的关键字放在箭头旁边用来标识打算引入的包是哪种类型的。预定义关键字《import》是为公共包引入使用的,而关键字《access》是为私有的包引入使用的。

表示选项

作为可选的箭头选项,它可以通过使用文本来表示一个元素引入,该文本在下面的括号中惟一地标识了被引入的元素。文本记法如下:

{import<qualifiedName>}or{access<qualifiedName>}

示例

在图 92 中,列出了一些包引入。在 Types 中的元素被引入到 ShoppingCart 中,然后又进一步被引入到 WebShop 中。然而,在 Auxiliary 中的元素只能从 ShoppingCart 中访问,不能通过使用来自 WebShop 中的没有量化的名称来引用它们。

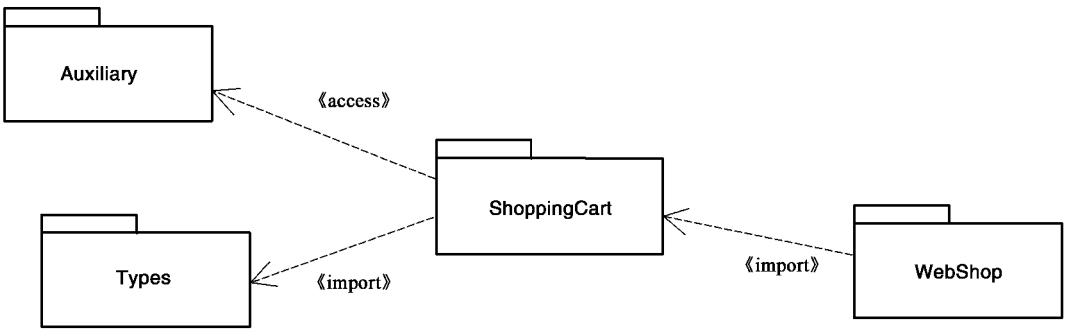


图 92 公有包和私有包引入的例子

9.7 操作图(Operations diagram)

构造包的操作图对行为特征、操作,以及参数构造进行了规格说明。如图 93 所示。

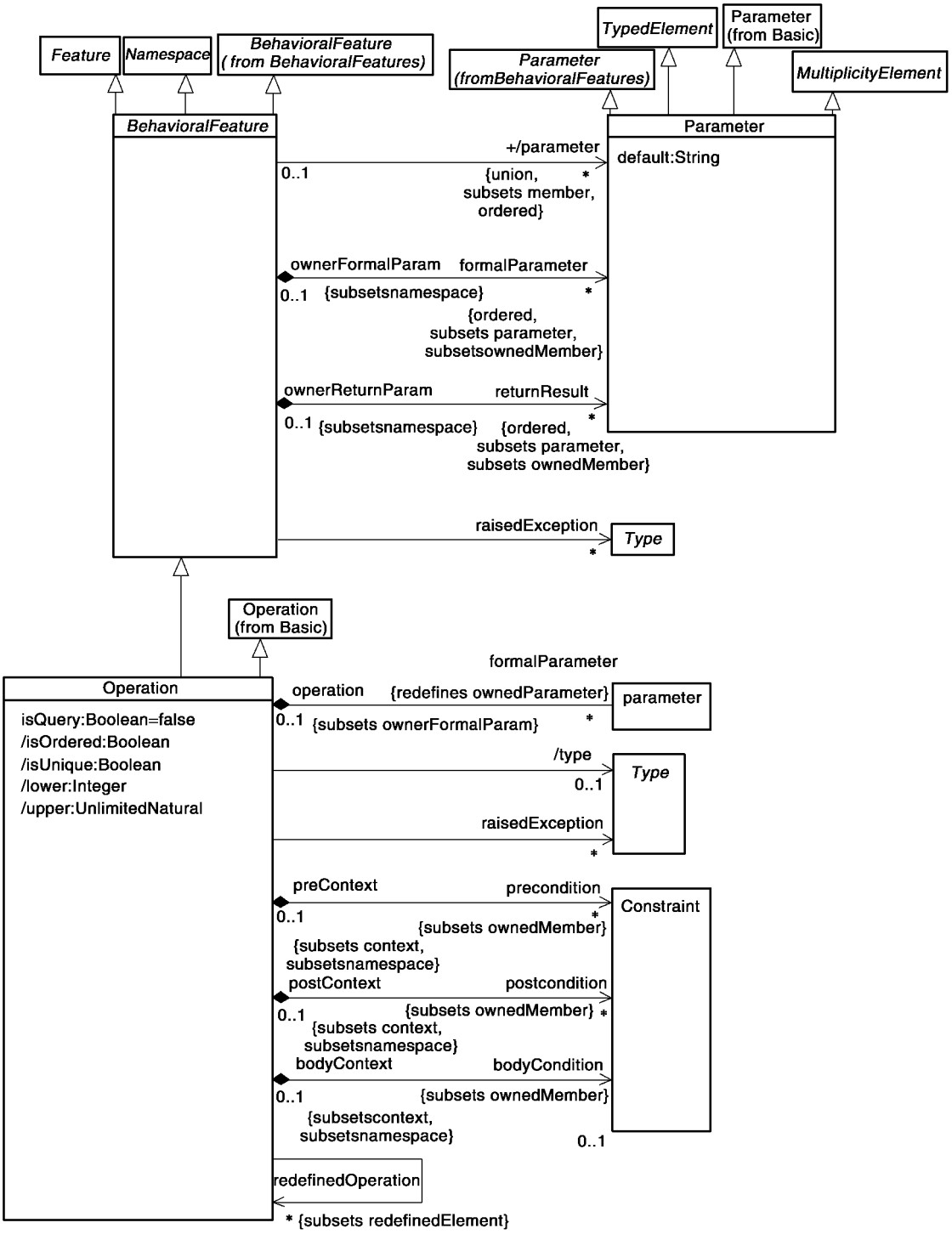

Feature
Namespace
BehavioralFeature
(from BehavioralFeatures)
TypedElement
Parameter
(from Basic)
Parameter
(fromBehavioralFeatures)
MultiplicityElement
BehavioralFeature
Parameter
default:String
+/parameter
0..1
{union,
subsets member,
ordered}
*
ownerFormalParam
formalParameter
0..1
{subsetsnamespace}
*
{ordered,
subsets parameter,
subsetsownedMember}
ownerReturnParam
returnResult
0..1
{subsetsnamespace}
{ordered,
subsets parameter,
subsets ownedMember}
raisedException
Type
Operation
(from Basic)
formalParameter
Operation
operation
{redefines ownedParameter}
parameter
isQuery:Boolean=false
/isOrdered:Boolean
/isUnique:Boolean
/lower:Integer
/upper:UnlimitedNatural
0..1
{subsets ownerFormalParam}
/type
Type
raisedException
preContext
precondition
Constraint
{subsets ownedMember}
{subsets context,
subsetsnamespace}
postContext
postcondition
{subsets ownedMember}
{subsets context,
subsetsnamespace}
bodyContext
bodyCondition
{subsets ownedMember}
{subsetscontext,
subsetsnamespace}
redefinedOperation
* {subsets redefinedElement}

图 93　构造包的操作图

9.7.1 行为特征[BehavioralFeature(特化的)]

描述

Constructs::BehavioralFeature——重用了来自 Abstraction::BehavioralFeature 的行为特征的定义。它向 Constructs::Namespace 和 Constructs::Feature 中添加了规格说明。

属性

无附加属性。

关联

a) 形参:Parameter[*]——定义该行为特征形参的有序规格集。子集:BehavioralFeature::parameter 和 Namespace::ownedMember。

b) 抛出例外:Type[*]——引用描述该特征激活过程中可能出现的例外的类型。

c) 返回值:Parameter[*]——定义该行为特征返回值的有序规格集。子集:BehavioralFeature::parameter 和 Namespace::ownedMember。

约束

无附加约束。

附加操作

[1] 查询 isDistinguishableFrom()决定了两个行为特征是否能共存于同一命名空间。它定义二者应具有不同的特征标记。

```
BehavioralFeature::isDistinguishableFrom(n:NamedElement,ns:Namespace):Boolean;
isDistinguishableFrom=
  if n.oclIsKindOf(BehavioralFeature)
  then
    if ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->notEmpty()
    then Set{}->include(self)->include(n)->isUnique(bf|bf.parameter->collect(type))
    else true
endif
else true
endif
```

语义

形参定义了调用行为特征时应提供的变元的类型和数目。返回值定义调用成功后返回的变元类型和数目。行为特征在其被调用过程中可能抛出例外。

记法

无附加记法。

9.7.2 操作[Operation(特化的)]

操作是类目的行为特征,它详细说明调用关联行为所需的名称、类型、参数及约束。

描述

Constructs::Operation 重用了基本包中的操作定义。它向 Constructs::BehavioralFeature 中添加了规格说明。

操作的规格说明定义了它所提供的服务,而非具体的内部实现,并包括前置、后置条件的列表。

属性

a) /isOrdered:Boolean——重定义来自 Basic 的相应性质,以从对应于本操作的返回值中获取这些信息。

b) /isQuery:Boolean——规格说明行为特征的执行是否不会改变系统状态(isQuery=true),或是否产生副作用(isQuery=false)。默认值为 false。

c) /isUnique:Boolean——重定义来自 Basic 的相应性质,以从对应于本操作的返回值中获取这些信息。

d) /lower:Integer[0..1]——重定义来自 Basic 的相应性质,以从对应于本操作的返回值中获取这些信息。

e) /upper:UnlimitedNatural[0..1]——重定义来自 Basic 的相应性质,以从对应于本操作的返回值中获取这些信息。

关联

a) 主体条件:Constraint[0..1]——本操作调用的返回值之上的一个可选约束。子集 Namespace.ownedMember。

b) 形参:Parameter[*]——规格说明本操作的形式参数。重定义了 Basic::Operation.ownedParameter 和 BehavioralFeature.formalParameter。

c) 后置条件:Constraint[*]——规格说明操作完成时系统状态的约束的可选集合。子集 Namespace.ownedMember。

d) 前置条件:Constraint[*]——调用操作时系统状态上的约束的可选集合。子集 Namespace.ownedMember。

e) 例外:Type[*]——引用描述在调用本操作过程中可能出现的例外的类型。重定义了 Basic::Operation.raisedException 和 BehavioralFeature.raisedException。

f) 重定义操作:Operation[*]——引用由本操作定义的操作。子集 Redefinable-Element.redefinedElement。

g) /type:Type[0..1]——重定义来自 Basic 的相应性质,以从对应于本操作的返回值中获取这些信息。

约束

[1] 若本操作仅有单一返回结果,isOrdered 等于该参数对应的 isOrdered 的值。否则 isOrdered 为 false。

isOrdered=**if** returnResult->size()=1 **then** returnResult->any().isOrdered **else** false **endif**

[2] 若本操作仅有单一返回结果,isUnique 等于该参数对应的 isUnique 的值。否则 isUnique 为 true。

isUnique=**if** returnResult->size()=1 **then** returnResult->any().isUnique **else** true **endif**

[3] 若本操作仅有单一返回结果,lower 等于该参数对应的 lower 的值。否则 lower 没有定义。

lower=**if** returnResult->size()=1 **then** returnResult->any().lower **else** Set{}**endif**

[4] 若本操作仅有单一返回结果,upper 等于该参数对应的 upper 的值,否则 upper 没有定义。

upper=**if** returnResult->size()=1 **then** returnResult->any().upper **else** Set{}**endif**

[5] 若本操作仅有单一返回结果,type 等于该参数对应的 type 的值,否则 type 没有定义。

type=**if** returnResult->size()=1 **then** returnResult->any().type **else** Set{}**endif**

[6] 只能为查询操作定义主体条件。

bodyCondition->notEmpty() **implies** isQuery

附加操作

[1] 查询 isConsistentWith()定义了语境的任何两个操作中,哪一个能够重定义,该重定义在逻辑上是否一致。如果一个重定义操作与另一个重定义操作有相同的形参数、相同的返回结果数,且每个形参和返回结果的类型与相应的重定义的形参和返回结果的类型一致,即这两个操作保持一致。

```
Operation::isConsistentWith(redefinee:RedefinableElement):Boolean;
pre:redefinee.isRedefinitionContextValid(self)
isConsistentWith=(redefinee.oclIsKindOf(Operation) and
                  let op:Operation=redefinee.oclAsType(Operation) in
                  self.formalParameter.size()=op.formalParameter.size() and
                  self.returnResult.size()=op.returnResult.size() and
                  forAll(i|op.formalParameter[i].type.conformsTo(self.formalParameter[i].
                  type)) and
                  forAll(i|op.returnResult[i].type.conformsTo(self.returnResult[i].type))
                  )
```

语义

操作在类目的实例中被调用，操作是类目实例的特征。

操作的前置条件定义在调用操作时应为 true 的条件。这些前置条件可由操作的执行来假定。

操作的后置条件定义在操作的调用成功完成时为 true 的条件，假定已满足前置条件。后置条件应由任意操作的实现来满足。

操作的主体条件约束了返回结果。主体条件与后置条件的区别在于，重新定义操作时，可重载主体条件，而只能添加后置条件。

操作在其调用过程中可能抛出例外。例外出现时，不能假定操作的后置条件或主体条件已得到满足。

操作可在特征化类目的特化作用中被重定义。该重定义限定形参或返回结果的类型，添加前置或后置条件，添加新出现的例外，或精化操作的规格说明。

每个操作都声明了它的应用是否会改变模型中的实例或任何其他元素的状态(isQuery)。

语义变化点

当前置条件不被满足时，操作调用的行为是一个语义变化点。

记法

操作作为如下形式的文本串显示：

visibility name(parameter-list):property-string

a) visibility 是操作的可见性。-visibility 可取消。

b) name 为操作名。

c) parameter-list 是以逗号分隔的形参列表，每一形参使用如下语法：

   name:type-expression[multiplicity]=default-value[{property-string}]

   1) **name** 为参数名。

   2) **type-expression** 确定参数类型。

   3) 方括号中的 **multiplicity** 是参数的势域——在假定条件[1]的情况下，可抑制 **multiplicity**。

   4) **default-value** 是对参数默认值的取值规格说明。**default-value** 为可选项(若省略 **default-value**，等号也可省略)。

   5) property-string 指出应用于参数的性质取值。**property-string** 为可选项(若没有定义任何性质，可省略大括号)。

property-string 随意地显示包括在大括号中的其余操作的性质。

表示选项

参数列表可省略。

样式指南

操作名以小写字母开头为特色。

示例

display()

－**hide**()

＋**createWindow**(**location**:**Coordinates**,**container**:**Container**[0..1]):**Window**

＋**toString**():**String**

9.7.3 参数[Parameter(特化的)]

参数是变元的规格说明,用于向行为特征的调用传递,或从中获取信息。

描述

Constructs::**Parameter** 集合了来自 **Basic** 和 **Abstractions**::**BehavioralFeatures** 的参数定义。它向类型化元素和多重性元素中加入规格说明。

参数是一种类型化元素,其目的为允许对参数上可选的势域的规格说明。另外,它支持可选默认值的规格说明。

属性

default:**String**[0..1]——定义一个字符串,在没有变元提供给参数时,由该字符串提供一个值。

关联

/操作:**Operation**[0..1]——引用操作,对该操作而言,这是一个形参。子集 **NamedElement**::**namespace** 并重定义 **Basic**::**Parameter**::**operation**。

约束

无附加约束。

语义

参数规格说明了如何向行为特征(如操作)的调用传入或获取变元。参数的类型和势域限定了传递哪些值,传递多少,以及这些值是否有序。

如果为参数定义默认值,将在调用时对其求值,当且仅当没有变元提供给行为特征的调用时,该默认值作为参数的变元使用。

记法

见“操作”。

9.8 包图(Packages diagram)

构造包的包图规格说明了包和包合并的结构。如图 94 所示。

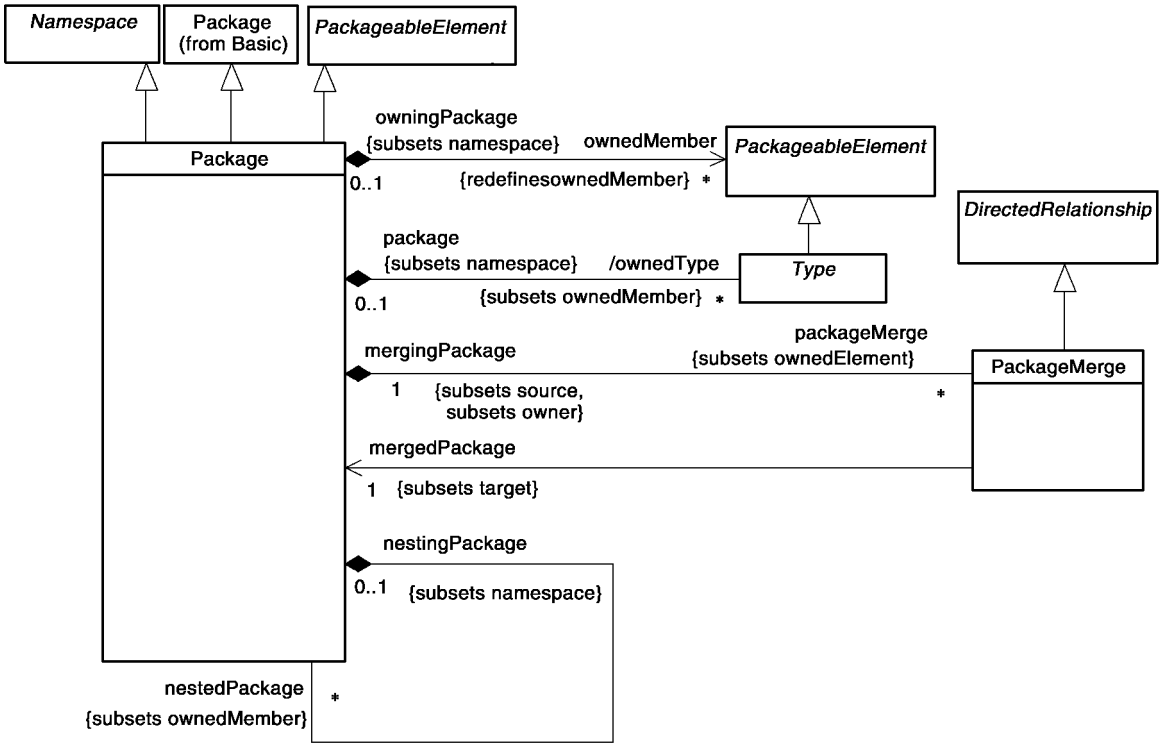


图 94　构造包的包图

9.8.1　类型[Type(附加性质)]

描述

Constructs::Type 在类目图中定义。包图添加了类型和包之间的关联,描述类型从属于包的所属关系。

属性

无附加属性。

关联

包:**Package**[0..1]——规格说明类目所属的包。子集 NamedElement::namespace 并重定义 **Basic::Type::package**。

约束

无附加约束。

语义

无附加语义。

9.8.2　包(Package)

包用于组织元素,并为这些组织在一起的元素提供一个命名空间。

描述

包是其成员的命名空间,且允许包含其他的包。仅有可封装的元素才能成为包的所有成员。利用命名空间的优势,包不但可引入其他包的个体成员,也可引入其他包的所有成员。

此外包还可为其他包所合并。

属性

无附加属性。

关联

a) 嵌套包：**Package**[*]——引用包的所有成员，该成员为包。子集 **Package::ownedMember**，并重定义 **Basic::Package::nested-Package**。

b) 所有成员：**PackageableElement**[*]——定义属于该包的成员。重定义了 **Namespace::owned-Member**。

c) 所有类型：**Type**[*]——引用包的所有成员，该成员为类型。子集 **Package::-wnedMember**，并重定义 **Basic::Package::ownedType**。

d) 包：**Package**[0..1]——引用包中拥有的包。子集 **NamedElement::namespace**，并重定义 **Basic::Package::nestingPackage**。

e) 包合并：**Package**[*]——引用属于该包的包合并。子集 **Element::ownedElement**。

约束

[1] 如果属于包的某个元素具有可见性，则它为 **public** 或 **private**。

self. ownedElements->forAll(e | e. visibility->notEmpty() implies e. visbility = # publie or e. visibility = # private)

附加操作

[1] 查询 **mustBeOwned**()指出该类型的元素是否应有一个所有者。

Package::mustBeOwned():Boolean

mustBeOwned = false

[2] 查询 **visibleMembers**()定义包中哪些成员可在外部被访问。

Package::visibleMembers():Set(PackageableElement);

visibleMembers = member->select(m|self. makesVisible(m))

[3] 查询 **makesVisible**()定义某个包是否使其成员在该包外部可见。无可见性和可见性为 **public** 的元素被定义为可见。

Package::makesVisible(el:Namespaces::NamedElement:Boolean;

pre:self. member->includes(el)

makesVisible = el. visibility->isEmpty()or el. visibility = # public

语义

包是一个命名空间，同时也是能够包含在其他包中的一个可封装元素。

在包中使用非限定名的元素是所有元素、引入元素以及封装(外部的)命名空间中的元素。每个所有元素和引入元素都可具有可见性，决定它们是否在包的外部可见。

包拥有它的所有成员，同时也暗示如果包从模型中移走，该包所拥有的元素也将被移走。通过使用限定名，包的公共内容在包外部总是可用的。

记法

包被表示为一个大矩形，在该大矩形顶部的左端附有一个小矩形("标签")。大矩形中可显示包的成员。成员也可以由画在包外面的、连接到成员元素的分支线来表示，在尾端画上中间是加号"+"的圆圈，附在命名空间(包)上。

a) 若大矩形中没有显示包的成员，则应在大矩形中置上包名；

b) 若大矩形中显示了包的成员，则应在标签中置上包名。

包元素的可见性可由元素名前的可见性记法(**public** 为'+'，**private** 为'-')来表示。

表示选项

某些工具通过图形绘制来显示可见性，如颜色和字体。某些工具也通过选择性地显示拥有定义可见性级别的元素来表明可见性，如仅显示 **public** 元素。描述包及其内容的图没有必要显示包的所有内容，它可根据某些标准显示元素的子集。

通过包引入或元素引入从而能够在引入它们的包中使用的元素，拥有明显的或暗淡的颜色，以表明它们是不能够被修改的。

示例

图 95 中对相同的包 **Types** 有三种记法。左边的图仅显示了包，而没有展现任何成员。中间的图在包的边界内显示了其部分成员；右边的这幅使用二选一的成员关系记法来显示部分成员。

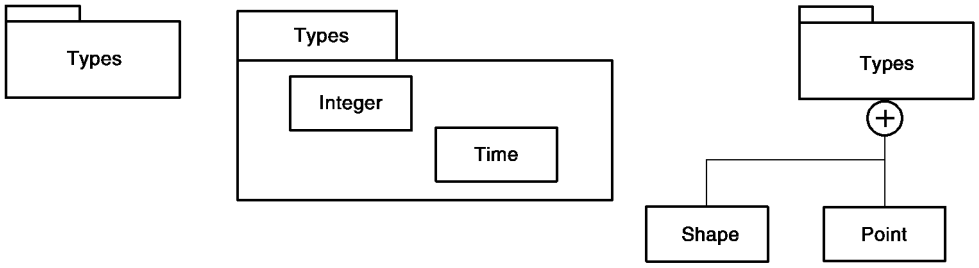


图 95　带成员的包的例子

9.8.3　包合并(PackageMerge)

包合并定义一个包如何通过合并它们的内容来扩展另一个包。

描述

包合并是两个包之间的关系，其中目标包(被指向的包)的内容通过规格说明和重定义，与源包的内容相合并。

当有意使用具有相同名称的元素来表征相同的概念，而不管它们在哪个包中定义时，采取以下机制：合并包可从一个或几个包中获取同名称同种类的元素，使用泛化和重定义将它们合并入一个单一的元素中。

应注意包合并可认为是明确定义泛化和重定义的捷径。被合并的包仍然可用，也能独立限定那些包中的元素。

从 **XMI** 的观点来看，既可将一个模型与所有保留的包合并交换，也可将模型与模型互换，其中所有的包合并都被转换掉(在这种情况下，使用包引入、泛化和重定义取而代之)。

属性

无附加属性。

关联

a) 被合并包：**Package**[1]——引用将被包合并的源包合并的包。子集：**DirectedRelat-ionship**::**target**。

b) 合并包：**Package**[1]——引用使用包合并的目标包的内容来进行扩展的包。子集：**Element**::**owner** 和 **DirectedRelationship**::**source**。

约束

无附加约束。

语义

两包之间的包合并意味着一个转化集，其中被合并包的内容在合并包中得到扩展。每个元素有它们自己特定的扩展规则。包合并被转化成包引入，它有着与包合并相同的源包和目标包。

包合并时，可见性为 **private** 的元素不在合并包中进行扩展。该规则递归地应用于被合并包的全部所有元素。

目标包的类目将转化成在源包中的同名类目，除非源包中已存在同种类同名称的类目。对前者，新类目获得对目标包类目的泛化。对后者，已存在的类目获得对目标包类目的泛化。在这两种情况下，所有的类型都引用被转化的类目，以这种方式在特殊类目中对常规类目的每个特征进行重定义。另外，源包的类目获得对目标包类目的每个被转化超类目的泛化。这是因为超类目可能已经合并入源包的附加性质中，这些性质需要适当地传播给类目。来自多重目标包的同种类同名称的类目被转化成源包的一个单一类目，包括对每个目标包类目的泛化。嵌套的类目也以同样的方式递归地进行转化。如果来自多重类目的特征不知何故发生了冲突，使用与多重继承相同的规则来解决这些冲突。

注意：从源包类目到目标包的同种类同名称类目的显式泛化是多余的，因为它将作为转化的一部分来产生。

目标（被合并）包的子包被转化成源（合并）包中的同名子包，除非源包已含有这样的同名子包。对前者，新的子包获得对目标包的子包的包合并。对后者，已经存在的包获得对目标包的子包的包合并。来自多重目标包的同名子包转化成源包中的一个单一子包，包括对每个目标子包的包合并。嵌套的子包以相同的方式递归地进行转化。

目标包的包引入转化成源包中相应的新的包引入。来自被引入包的元素不进行合并（除非也有一个对被引入包的包合并）。被合并元素的名称优先于被引入元素的名称，意味着在命名冲突的情况下，将隐藏被引入元素的名称，并需要参考使用限定符。目标包的元素引入转化成源包中相应的新的元素引入。被引入元素不进行合并（除非也有一个对拥有被引入元素或其别名的包的包合并）。

目标包的不可泛化可封装元素被拷贝到源包中。如果转化类目存在，则任何作为可封装元素的一部分被引用的类目，被重定向到转化的类目。

记法

包合并使用一条带箭头的虚线来表示，箭头从合并（源）包指向被合并（目标）包。另外，在虚线附近显示关键字“**merge**”。如图 96 所示。

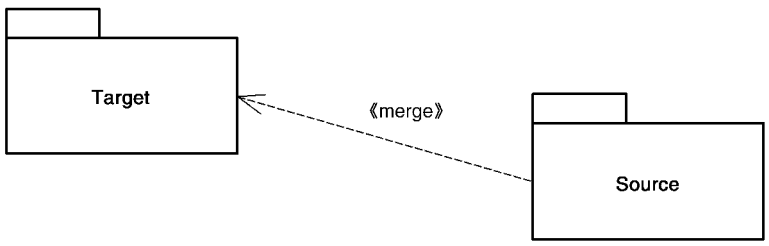


图 96　包合并的记法

示例

在图 97 中，包 R 分类合并了包 P 和包 Q，而包 S 只合并了包 Q。

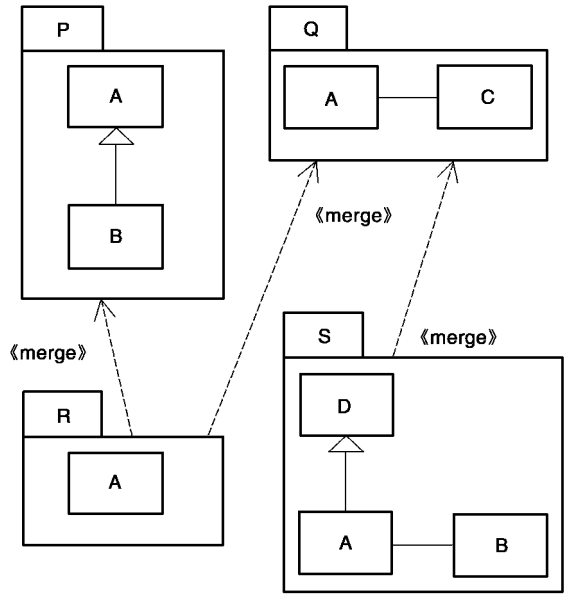


图 97　包合并的简单例子

图 98 展示了转换后的包 R 和包 S。尽管图中没有表示出来，但实际上包的分类合并已经转变为包的引入。

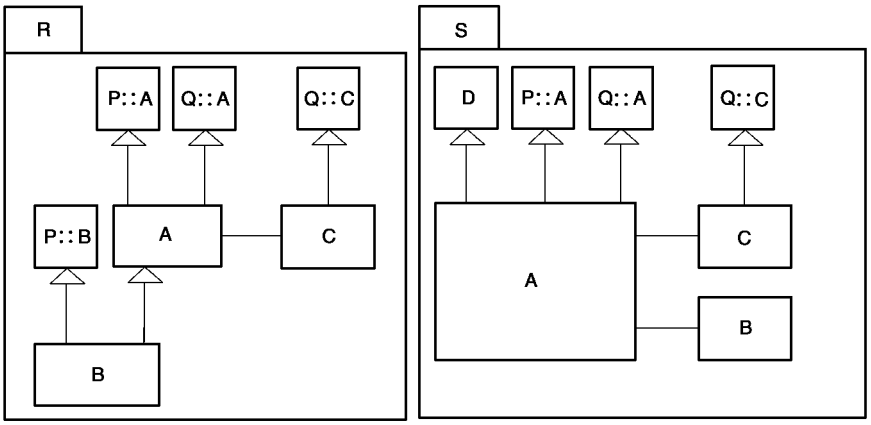


图 98　转化后的包的简单例子

图 99 对附加的包的分类合并作了介绍，它使包 T 分类合并了先前定义的包 R 和包 S。合并之前，包 T 完全为空。

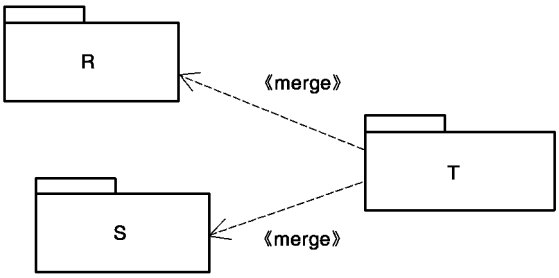


图 99　引入附加包合并

图 100 中描述了转化后包 T 的形式。在这个包中，A、B、C 和 D 的局部定义都集合在一起。包的合并仍然转变为包的引入。注意原先在包 Q 和包 S 中的关联端的类型都被更新成为包 T 中适用的类型。

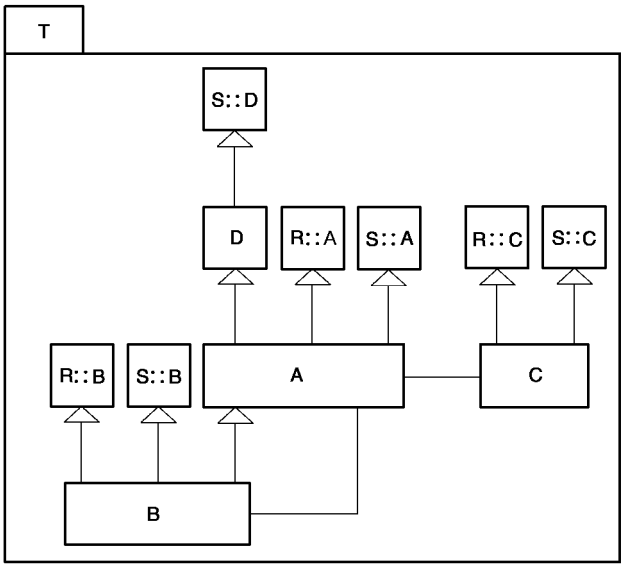


图 100　附加的包合并的结果

可以略去每一类目的不是所有但是大部分的细节，得到包合并转换最终结果的更清楚描述，如图 101 所示。

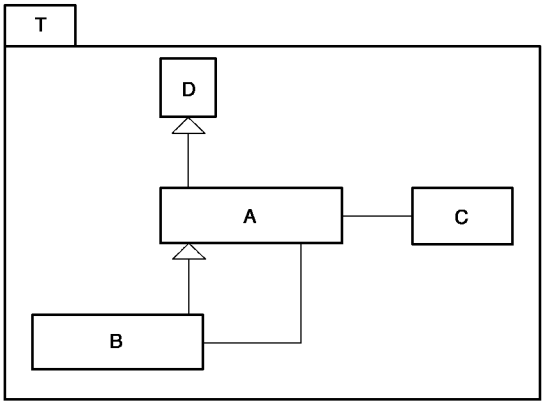


图 101　附加的包合并的结果:略图

10　核心::原子类型(Core::PrimitiveTypes)

基础构造库核心包中的原子类型包包含了大量定义元模型的抽象语法时使用的预定义类型。如图 102 所示。

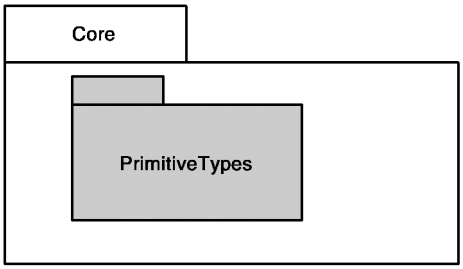


图 102　核心包由基础构造库包拥有且包含若干子包

10.1　原子类型包

核心包内的原子类型子包定义了原值的不同类型,原值用于定义核心元模型,并打算让每一个基于核心的元模型重用以下原子类型。

在核心和 UML 元模型中,预定义了这些原子类型,它们随时可为核心和 UML 扩展所用。这些预定义的值类型独立于所有对象模型和部分的核心定义。如图 103 所示。

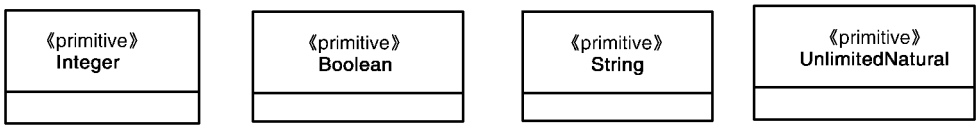


图 103　原子类型包中定义的类

10.1.1 布尔型(Boolean)

布尔类型用于逻辑表达式,由预定义的值 true 和 false 组成。

描述

布尔型是原子类型的一个实例。在元模型中,布尔定义了表示逻辑条件的枚举。枚举值为:

a) true——满足布尔条件;

b) false——不满足布尔条件。

用于元模型中的布尔属性和布尔表达式,如 OCL 表达式。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

布尔是原子类型的一个实例。

记法

布尔在元模型中作为属性类型出现。布尔实例是与 slot 相关联的值,可以拥有下列取值:true,或 false。

示例

如图 104 所示。

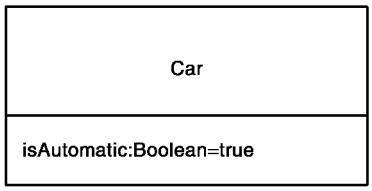


图 104 布尔属性的例子

10.1.2 整型(Integer)

整型是描述整数值的一种原子类型。

描述

整型的一个实例是(无穷)整数集合(…,−2,−1,0,1,2,…)中的一个元素。它用于元模型中的整型属性和整型表达式。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

整型是原子类型的一个实例。

记法

整型在元模型中作为属性类型出现。整型实例是与 slot 关联的值，如 1，－5，2，34，26524，等。

示例

如图 105 所示。

图 105　整型属性的例子

10.1.3　串(String)

串是某些合法字符集合中字符的序列，用于显示模型的信息。字符集合可以包含非罗马字母和字符。

描述

一个串的实例定义了一小段文本。串自身的语义依赖于它的目的，可以是一条注释，计算语言表达式、OCL 表达式等。串用于元模型中的串属性和串表达式。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

串是原子类型的一个实例。

记法

串在元模型中作为属性类型出现。串实例是与 slot 关联的值。该值是包括在双引号(“”)中的字符序列。假定基本字符集足够大，可描述多样的人类语言中的多字节字符；特别地，传统的 8 位 ASCII 字符集不能满足需要。假定计算机和应用程序能正确地对串进行操作和存储，包括避开为特殊字符制定的规范，该文档假定可以使用任意串。

串被作为文本串图形来显示。常规的可打印字符能够直接显示。不能打印字符的显示未给出详细说明，它们与平台相关。

示例

如图 106 所示。

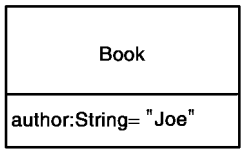


图 106　串属性的例子

10.1.4 不受限自然数(UnlimitedNatural)

不受限自然数是描述不受限自然数值的原子类型。

描述

不受限自然数的一个实例是(无穷)自然数集(0,1,2,…)中的一个元素。无穷值用星号(*)表示。

属性

无附加属性。

关联

无附加关联。

约束

无附加约束。

语义

无限自然数是原子类型的一个实例。

记法

在元模型中,无限自然数作为上限的类型出现。无限自然数实例是与 slot 关联的值,如 1,5,398 475 等。无穷值可用星号(*)表示。

示例

如图 107 所示。

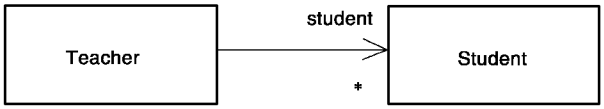


图 107 无限自然数的例子

11 核心::外廓(Core::Profiles)

基础构造库的外廓包中包含一些机制,这些机制允许扩展现有元模型中的元类,使它们适应不同的用途。这包括使 UML 元模型适应不同平台(如 J2EE 或.NET)和领域(如实时或企业过程建模)。外廓机制与 OMG 元对象工具(MOF)相容。

可扩展性

外廓机制不是一等扩展机制,它不允许修改现有元模型。确切地说,外廓的目的是提供一种直截了当的机制,通过定义针对特殊领域、平台或方法的结构来扩展已有的元模型。外廓中聚合了所有这样的改进。使用外廓不能消除任何应用于元模型(如 UML)的约束,但是可以添加特定的新约束。仅有的其他限制是外廓机制中固有的,且不会限制客户化元模型的方式。

MOF 中对一等可扩展性进行了处理,它没有限制允许对元模型进行哪些操作:若有必要,可以添加或移除元类和联系。当然也可以加以强行约束,不允许修改,只能扩展现有元模型。这样,一类可扩展性机制与外廓开始接合(当然也可以加入方法学的限制)。

打算定制元模型出于几种原因:

a) 提供适合特殊平台或领域的术语(例如获得 EJB 术语,如主接口、企业版 **java beans**、**Archive**);

b) 提供无记法结构的语法(例如在有动作的情况下);

c) 提供现有记法的另一种不同记法(例如可以使用一幅计算机的图片取代通常的结点记法,来描述网络上的一台计算机);

d) 添加元模型中尚未详细说明的语义(如在状态机中接受到信号时如何处理优先级);

e) 添加元模型中不存在的语义(如定义定时器、时钟或持续时间);

f) 添加限制元模型的使用方式及其构造的约束(如不允许单一转移中的并发操作);

g) 添加在将一模型转换成另一模型或代码时使用的信息(如定义模型和 JAVA 代码之间的映射法则)。

外廓和元模型

何时应该创建新的元模型,何时又应创建新外廓,没有简单的答案能将这个问题阐明。

11.1 外廓包(Profiles package)

外廓包依赖于来自核心的构造包,如图 108 所示。

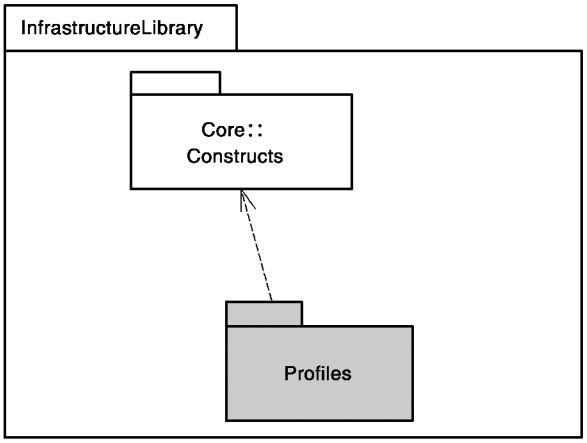


图 108 外廓由基础构造库包所拥有

外廓包中的类如图 109 所示,在后文中还将有详细描述。

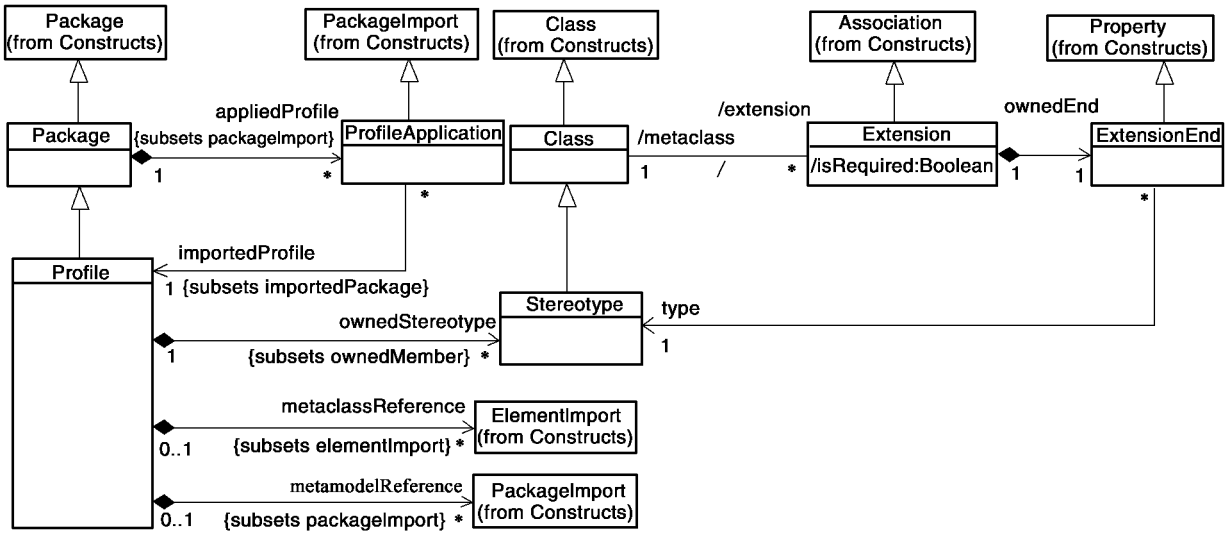


图 109 外廓包中定义的类

11.1.1 扩展(自外廓)

扩展用于表示通过衍型扩充元类的性质,并提供了在类中灵活添加(及后文说明的移除)衍型的能力。

描述

扩展是关联的一种。它的一端为平常性质,另一端为扩展端。前者将扩展绑定在类上,后者将扩展绑定在用于扩展类的衍型上。

属性

/**isRequired**:**Boolean**——表示在扩展类的实例建立时,是否应建立扩展版类型的实例。该属性值源于 **Extension**::**ownedEnd** 的势域;势域为 1 表明 **isRequired** 的值为 **true**,否则为 **false**。扩展端的默认势域为 0..1,因此 **isRequired** 的默认值为 **false**。

关联

a) **ownedEnd**::**ExtensionEnd**[1]——引用由衍型定义类型的扩展端。重新定义了 **Association**::**ownedEnd**。

b) /元类:**Class**[1]——引用被扩展的类。其性质源于非 **ownedEnd** 的 **menberEnd** 的类型。

约束

[1] 扩展的非所属端由 Class 定义类型。

metaclassEnd()->**notEmpty**()**andmetaclass**()->**ocllsKindOf**(**Class**)

[2] 扩展是二元的,它仅有两个 **menberEnd**。

self. menberEnd->size()=2

附加操作

[1] **metaclassEnd**()返回由元类确定类型的 **Property**(相对于衍型)。

Extension::**metaclassEnd**():**Property**

metaclassEnd=**menberEnd->reject**(**ownedEnd**)

[2] **metaclass**()返回被扩展的元类(相对于扩展的衍型)。

Extension::**metaclass**():**Class**;

metaclass=**metaclassEnd**(). **type**

[3] 如果所属端拥有下限为 1 的势域,**isRequired**()为 **true**。

Extension::**isRequired**():**Boolean**;

isRequired=(**ownedEnd->lowerBound**()=1)

语义

必需的扩展意味着衍型的实例应始终链接到被扩展的元类的实例上。仅当删除被扩展元类的实例,或从包的应用外廓中移除外廓定义的衍型时,才能显式删除衍型的实例。

当 **isRequired** 为 **true** 时,若没有衍型的实例,模型就不是良构的。

非必需的扩展意味着衍型的实例可以任意链接到被扩展元类的实例上,随后也可以任意删除。但是不要求对每个元类的实例进行扩展。当删除扩展元类的实例,或从包的应用外廓中移除外廓定义的衍型时,进一步删除衍型的实例。

为了在描述约束时能操纵被扩展的元类,端应有一个名称。若没有取名,缺省名为 **baseClass**。

记法

表示扩展的记法是从衍型指向扩展类的箭头,指向扩展类的一头是一个填充的三角形。扩展可以拥有与普通关联相同的修饰,但不会显示导航性箭头。如果 **isRequired** 为 **true**,在靠近扩展端的位置将显示性质{**required**}。如图 110 所示。

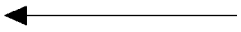


图 110　对扩展的记法

表示选项

可以在扩展端使用势域 0..1 或 1,作为性质{**required**}的替换。由于关系到 **isRequried** 的起源,势域 0..1 对应 **isRequired** 为 **false**。

样式指南

扩展的修饰被显式省略。

示例

图 111 中,显示了一个使用扩展的简单例子,其中的衍型 **Home** 对元类 **Interface** 进行了扩展。

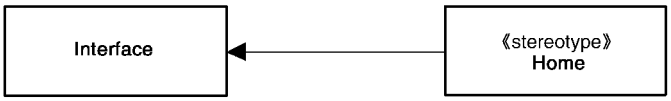


图 111　采用扩展的例子

可在类 **Interface** 的实例中任意添加或删除衍型 **Home** 的实例。从而提供一种灵活的方法,能在模型中动态添加或移除针对外廓的特定信息。

在图 112 中,由于定义了扩展为必需,衍型 **Bean** 的实例应始终链接在类 **Component** 的实例上。(因衍型 **Bean** 是抽象的,这表示其某个具体子类的实例应始终链接到类 **Component** 的实例上)。若不应用这种衍型,就不能很好地构造模型。这样规定了一种方法,它描述了扩展应该始终针对所有基础元类的实例出现,这些基础元类依赖于所应用的外廓。

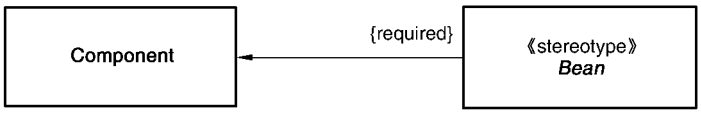


图 112　所需扩展的例子

注:对 UML1.4 的更改 UML1.x 中,扩展不作为元类存在。UML1.4 中 Stereotype::baseClass 的出现被映射到扩展的实例上。其中衍型确定所属端的类型,元类确定其他端的类型,元类又由基类指出。

11.1.2　扩展端(自外廓)

扩展端用于在扩展元类时将扩展绑定在衍型上。

描述

扩展端是一种通常由衍型确定类型的性质。

扩展端不具有导航性。如果它具有导航性,将成为扩展类目的性质。由于不允许外廓改变引用的元模型,所以不能在扩展类目中添加性质。因而扩展端仅为扩展所特有。

扩展端的聚合关系是组合关系。

扩展端的默认势域为 0..1。

属性

无附加属性。

关联

类型:衍型[1]——引用扩展端的类型。注意这种关联限制了扩展端的类型只能是衍型。重定义了**Property::type**。

约束

[1] 扩展端的势域为 0..1 或 1。

(self->lowerBound()=0 or self->lowerBound()=1) and self->upperBound()=1

[2] 扩展端的聚合关系是组合关系。

self.aggregation=#composite

附加操作

[1] **lowerBound()**以整数的形式返回势域的下限。这是对默认下限 1 的重新定义。

ExtensionEnd::lowerBound():[integer];

lowerBound=iflowerValue->isEmpty() then 0 else lowerValue->integerValue() endif

语义

无附加语义。

记法

无附加记法。

示例

见“扩展(来自外廓)”。

注:对 UML1.4 的改变 UML1.4 中扩展端不作为元类存在。更多细节见“扩展(来自外廓)”。

11.1.3 类(自构造,外廓)

描述

类具有导出关联,关联指出了如何通过一个或几个衍型对类进行扩展。

一个衍型是一个类,因此衍型不仅可应用于类,也可应用于衍型的定义。

属性

无附加属性。

关联

扩展:**Extension**[*]——引用详细说明元类附加性质的扩展。性质源于成员端的类型由类确定的扩展。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

表示选项

由衍型扩展的类具有可选择的关键字“**metaclass**”,该关键字在类名前或类名上方显示。

示例

图 113 中给出的例子清楚表明了扩展类 **Interface** 事实上是一个元类(来自引用元模型)。

图 113　对已扩展的类为元类的图示

注：对 UML1.4 的更改由 UML1.4ModelElement::stereotype 确定类型的链接被映射到由 Class::extension 确定类型的链接上。

11.1.4　包(自构造,外廓)

描述

包拥有一个或多个外廓应用程序,以表明使用了哪些外廓。

一个外廓是一个包,因此外廓不仅可应用于包,也可应用于外廓。

属性

无附加属性。

关联

应用外廓:ProfileApplication[*]——引用外廓应用程序,指明对包应用了哪些外廓,子集 Package::packageImport。

约束

无附加约束。

语义

无附加语义。

记法

无附加记法。

注：对 UML1.4 的更改 UML1.x 中无法指出对包应用了哪些外廓。

11.1.5　外廓(自外廓)

外廓定义了对引用元模型的有限扩展,目的是使元模型适用于特定的平台或领域。

描述

外廓是一种扩展引用元模型的包。主要的扩展构造是衍型,衍型被定义为外廓的一部分。

外廓通过使用外廓包中定义的元类,在普通的元模型建模中引入了一些约束,或限制。

外廓是元模型的受限形式,如下文所述,它应始终与引用元模型相关,如 **UML**。

外廓没有引用元模型则无法使用,同时它定义了有限的能力来扩展引用元模型的元类。扩展被定义为应用于现有元类的衍型。

属性

无附加属性。

关联

a)　元类引用:**ElementImport**[*]——引用可扩展的元类。**Package::elementImport** 的子集。

b)　元模型引用:**PackageImport**[*]——引用包含(直接或间接地)可扩展元类的包。**Package::packageImport** 的子集。

c)　所有衍型:**Stereotype**[*]——引用属于外廓的衍型。**Package::ownedMember** 的子集。

约束

[1] 外廓中，作为元类引用引入的元素没有被特殊化或泛化。

```
self. metaclassReference. importedElement->
  select(c|c. oclIsKindOf(Classifier)and
  (c. generalization. namespace=selfor
  (c. specialization. namespace=self))->isEmpty()
```

[2] 所有作为元类引用引入，或者通过元模型引用引入的元素，都是相同的基础引用元模型的成员。

```
    self. metamodelReference. importedPackage. elementImport. importedElement.
  allOwningPackages()->union(self. metaclassReference. importedElement.
  allOwningPackages())->notEmpty()
```

附加操作

[1] **allOwningPackages()**返回所有直接或间接拥有的包。

```
NamedElement::allOwningPackages():Set(Package)
allOwningPackages=self. namespace->select(p|p. oclIsKindOf(Package))->
  union(p. allOwningPackages())
```

语义

外廓通过定义来扩展引用元模型或其他的外廓。定义一个独立的外廓是无法直接或间接扩展现有元模型。外廓机制可以与任何从 **MOF** 创建的元模型，包括 **UML**、**CWM** 一起使用。

引用元模型由引入的或本地拥有的元类组成，这是它的特点。所有由同一外廓扩展的元类应是相同的引用元模型的成员。哪些元类被扩展的相关信息，能被工具用于多种用途。例如，在使用外廓时，进行元素的过滤和隐藏，又如提供应用于特定外廓的特殊工具条等。然而，不能简单地通过使用外廓来删除包或模型的元素。因此，每个外廓都提供了一种简单的视图机制。

作为外廓的一部分，衍型之间或衍型与元类之间不能存在关联，除非它们是引用元模型中现存关联的子集。但是，普通类之间，以及衍型到普通类可以存在关联。同样地，元类或衍型不能确定衍型的性质的类型。

记法

外廓使用与包相同的记法，同时增加了关键字“**profile**”，该关键字在外廓名前或其上方出现。**Profile::metaclassReference** 和 **Profile::metamodelReference** 分别使用与 **Package::elementImport** 和 **Package::packageImport** 相同的记法。

示例

图 114 中，显示了 EJB 外廓的一个简单例子。

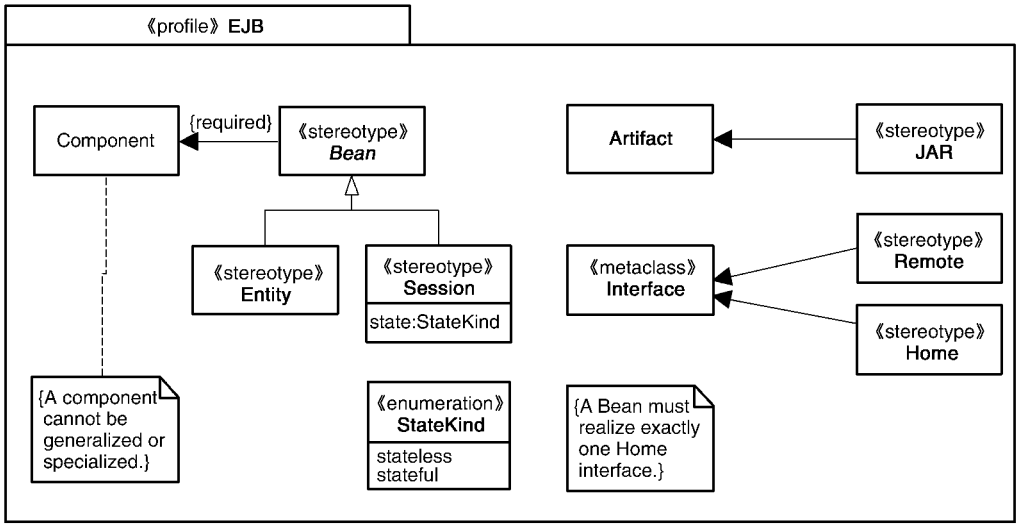


图 114　定义简单的 EJB 外廓

该外廓规定抽象衍型 **Bean** 必须应用于元类 **Component**，即表明 **Bean** 的具体子类 **Entity** 和 **Session** 的实例都应链接到 **Component** 的每个实例上。该约束是外廓的一部分，当对包应用外廓时，对其进行评价。需要满足该约束以便合法地组成模型。

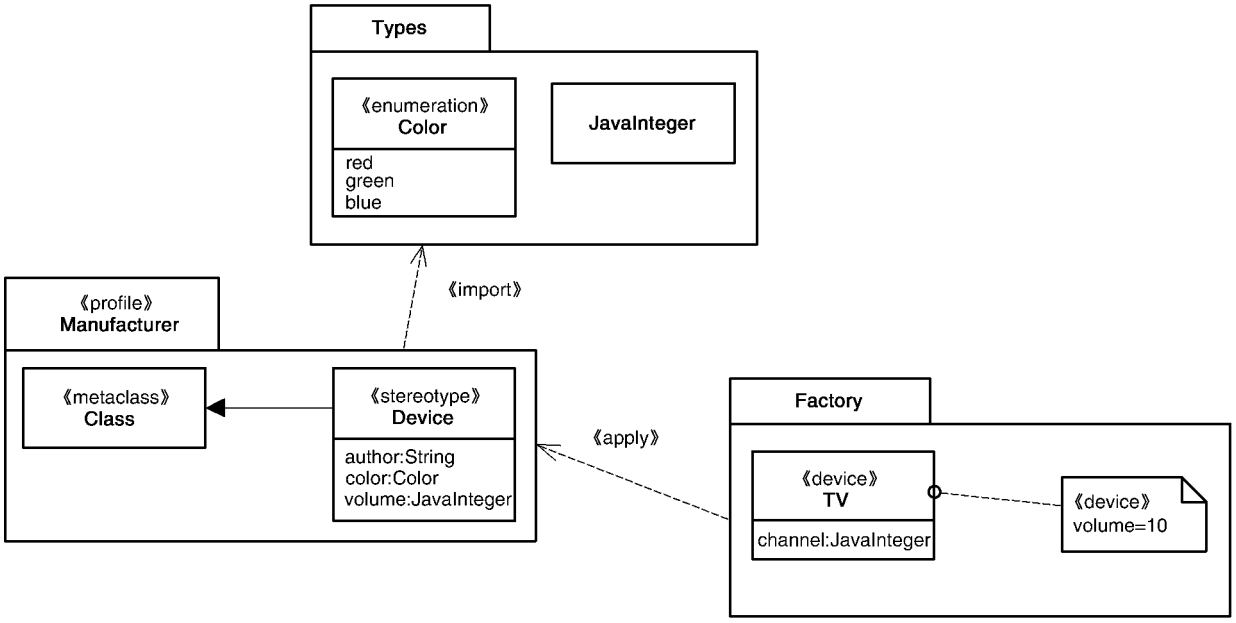


图 115　从外廓引入包

图 115 中，在外廓 **Manufacturer** 中引入了包 **Types**。数据类型 **Color** 作为衍型 **Device** 的某一性质的类型使用，正如同时也使用了预定义的类型 **String**。注意类 **JavaInteger** 也可作为性质的类型使用。

如果稍后对某包应用外廓 **Manufacturer**，则 **Types** 的类型在该包中也是可用的（因为外廓的应用是一种引入）。这意味着，举例说明，类 **JavaInteger** 既能作为元性质（衍型 **Device** 的一部分），又能作为普通性质（类 **TV** 的一部分）使用。注意对类 **TV** 应用衍型 **Device** 时，是如何对元性质赋值的。

11.1.6 外廓应用(自外廓)

外廓应用用于显示对包应用了哪些外廓。

描述

外廓应用是包引入的一种,它增强性能以表明对包应用了外廓。

属性

无附加属性。

关联

引入外廓:**Profile**[1]——外廓应用过程中应用于该包的外廓。子集 **Package::Import::importedPackage**。

约束

无附加约束。

语义

可任意将一个或几个外廓应用于某个包,该包创建自外廓扩展的元模型。应用外廓意味着允许但不是必须应用衍型,该衍型被定义为外廓的一部分。可以对包应用多重外廓,直到它们出现冲突的约束为止。如果某外廓的应用依赖于其他外廓,则应首先应用其他外廓。

当应用外廓时,应该为某些元素创建适当的衍型实例,这些元素是必须扩展的元类的实例。若没有这些衍型实例,将不能很好地组成模型。

一旦对包使用了外廓,允许任意移走应用的外廓。移走外廓意味着删除外廓中定义的元素的所有实例。不能移走已应用的外廓,除非首先移走依赖于该外廓的其他应用外廓。

移走应用外廓使引用元模型的元素实例保持完整。仅仅删除外廓的元素实例。这表示,例如,一个外廓 **UML** 模型总能与另一个不支持该外廓的工具互换,并可被解释为一个纯粹的 **UML** 模型。

记法

使用带有箭头的虚线来表示外廓,箭头从包指向应用的外廓。箭头附近显示关键字"**apply**"。

如果多重应用外廓具有同名衍型,则可能需要限定衍型的名称(带有外廓名)。

示例

给定外廓 **Java** 和 **EJB**,图 116 说明了它们是如何应用到包 **WebShopping** 上的。

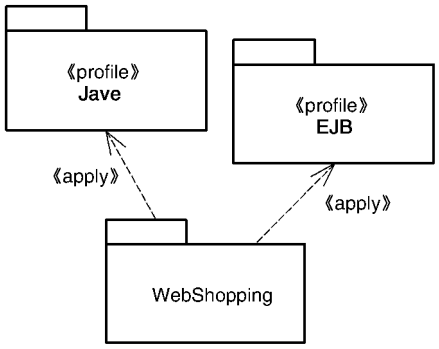


图 116 适用于包的外廓

11.1.7 衍型(自外廓)

衍型定义如何扩展现有元类(或衍型),以及这些被扩展的元类在不同平台或领域中使用的专门术语和记法。

描述

衍型是类的一种，通过扩展对类进行扩展。

正如类一样，衍型也可以有性质，性质作为标志定义来引用。对模型元素应用衍型时，性质的取值可以作为加标值引用。

属性

无附加属性。

关联

无附加关联。

约束

[1] 一个衍型可能仅仅泛化或特殊化另一个衍型。

self. generalization. general->forAll(e|e. oclIsKindOf(Stereotype))and

self. specialization. specific->forAll(e|e. oclIsKindOf(Stereotype))

[2] 具体的衍型应直接或间接地对类进行扩展。

not self. isAbstract **implies** self. extensionEnd->union(self. parents. extensionEnd)->notEmpty()

语义

衍型是一种受限元类，不能被自身使用，但应始终与它扩展的某个元类联合使用。

每个衍型可通过扩展扩展一个或多个类，将之作为外廓的一部分。与此类似，一个类也可被一个或多个衍型扩展。

衍型的实例被链接到扩展的元类(或衍型)的实例上，这借助了它们的类型之间的扩展。

记法

衍型使用与类相同的记法，另外在类名前或其上方显示关键字**"stereotype"**。

对模型元素应用某衍型时(衍型的实例链接到元类的实例)，该衍型的名称由一对书名号括起来，在衍型类名前或其上方显示。

如果应用了多重衍型，这些应用衍型的名称显示为一对双尖括号中的以逗号分隔的列表。

表示选项

若对某元素应用了多重衍型，可将每个衍型名放入一对双尖括号中，再将之一一列出。

应用于模型元素的衍型的值可作为附在该模型元素上的注释符的一部分来显示。来自特定衍型的值可选地先于应用衍型的名称，这些值用一对书名号括起来，在需要显示多个应用衍型的值时有用。

如果为扩展端定义了名称，当衍型应用于模型元素时，可在出现衍型名的地方，使用包括在一对书名号中的该扩展端名。

可为衍型附上特殊记法，在应用了衍型的模型元素的记法出现的地方使用。

样式指南

应用的衍型的首字母不能大写。通常不显示应用的衍型的值。

示例

图 117 中，定义一个简单的衍型 **Clock** 可任意(动态地)应用于元类 **Class** 的实例。

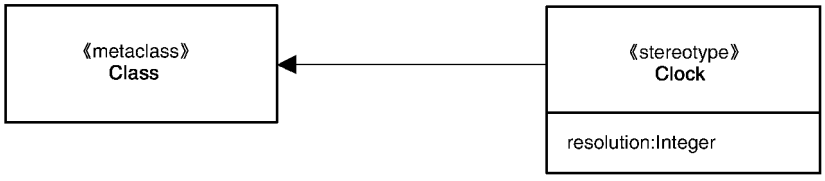


图 117 定义一个衍型

注意：为了在衍型 **Clock** 上写入约束，该约束能应用于元类 **Class** 或 **Class** 的任意一个关系，需要为由元类定义类型的端给出一个名称，以达到导航的目的。一个典型的这种名称为 **base**。

图 118 中，显示了图 117 中所举范例的一个实例规格说明。注意扩展应是复合的，且该例中导出的必需属性为 **false**。

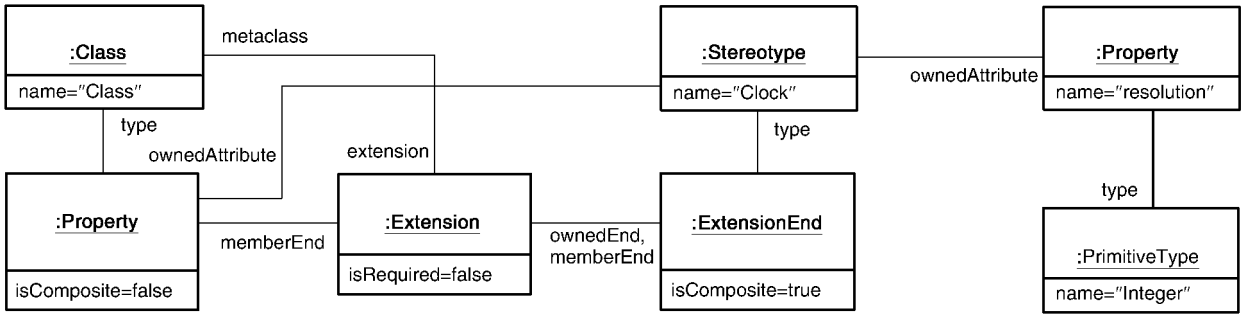


图 118　定义衍型时的实例规约

图 119 中，显示了同样的衍型 **Clock** 如何既能扩展元类 **Component**，又能扩展元类 **Class**。同时也显示了不同衍型如何扩展相同元类。

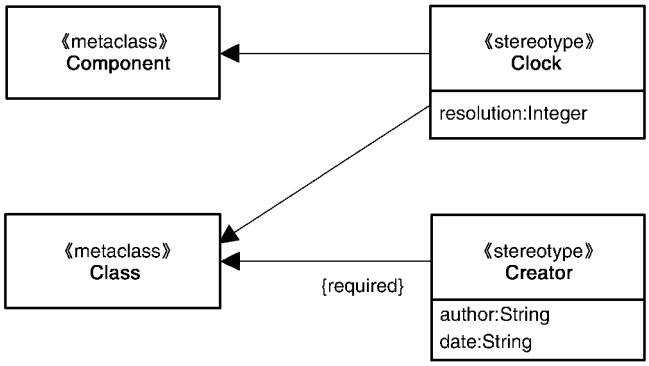


图 119　在多重衍型上定义多重衍型

图 120 显示了衍型 **Clock**，如图 119 所定义，是如何被应用于叫做 **StopWatch** 的类上的。

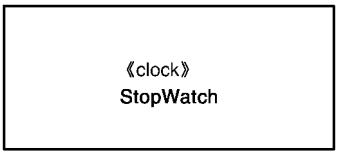


图 120　使用衍型

当衍型 **Clock** 被应用于叫做 **StopWatch** 的类上时，图 121 为之显示了一个实例模型。衍型与元类之间的扩展导致了衍型 **Clock** 的实例与（用户定义的）类 **StopWatch** 之间的链接。

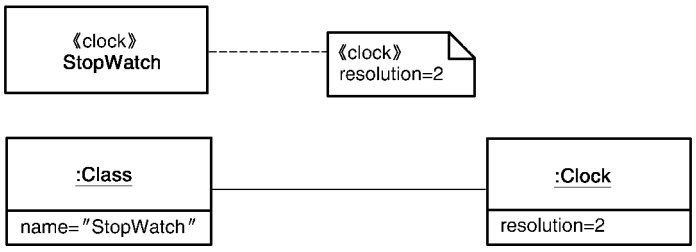


图 121　显示衍型的值及简单的实例规约

接下来，两个规约，Clock 和 Creator，被应用于同一模型元素，如图 122 所示。注意每个应用规约的属性取值可显示在附在模型元素上的注释符中。

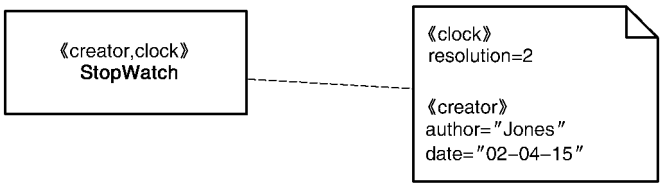


图 122　使用规约并图示值

中 华 人 民 共 和 国
国 家 标 准
统一建模语言(UML)
第1部分:基础结构
GB/T 28174.1—2011
*
中国标准出版社出版发行
北京市朝阳区和平里西街甲2号(100013)
北京市西城区三里河北街16号(100045)
网址:www.gb168.cn
服务热线:010-68522006
2012年9月第一版
*
书号:155066·1-45145

GB/T 28174.1-2011