Chapter 3

UML Infrastructure: Core Concepts

The definition of the second major release of UML 2 has been a long-awaited process. In 2000, the OMG issued four Requests for Proposals (RFP) for UML 2.0, Infrastructure, XML Metadata Interchange (XMI), and Superstructure. Most users are concerned only with the last Superstructure that contains modeling concepts having a direct relationship with their application blueprints. As this textbook is partly addressed to researchers, when necessary, we must expose the Infrastructure that contains basic constructs merged by the Superstructure. Some figures are reproduced as is from the standard. Others are synthesized, to explain this complex standard in an understandable way. Comments in quotes (") or in italic are extracted from the UML. For those who are interested in details about metamodeling, please consult Atkinson and Kühne (2002), Kobryn (2004), and Alanen and Porres (2005); the following description gives only a snapshot and an overview of the current UML version contents necessary to understand how the UML is built from its first stone. The rationale behind metamodeling proposed by the OMG is to create a family of modeling languages (UML is one of them) based upon a common core of concepts and to avoid a collection of unrelated specialized languages that could impair interoperability.

To explain the metaconcepts in the Infrastructure and the Superstructure, the UML makes use of graphical notations of itself. This kind of "bootstrapping" supposes that the reader who reads the Infrastructure Book is already familiar with the UML notation (class, association, aggregation/composition, inheritance/derivation, constraints, roles, multiplicities at the association endpoints, etc.) or mathematics like subsetting or derived union (Rundensteiner and Bic, 1992) to understand diagrams at the metalevel. That supposition does not stand for readers who want to learn UML for the first time. So, normally, the neophyte must start with the Superstructure Book (Chapter 4) without paying any attention to the abstract and the way modeling packages are tied together and tied to packages defined in the Infrastructure Book. Once the UML is fully understood, readers can reiterate through Infrastructure (Chapter 3) and Superstructure to strive to understand why he cannot draw, for instance, an association between the two object states ("not UML compliant"), but only dependencies.

D. M. Bui, Real Time Object Uniform Design Methodology with UML, 47–75.

Researchers interested in ontology engineering must read everything because UML is itself an example of ontology.

(Note: If there is some difficulty in interpreting the diagrams of this Infrastructure part, do not be frustrated, have a quick look at the kind of diagrams and concepts developed in the Infrastructure, then skip this arid, hermetic, and highly technical description because some symbols used are in fact described later when studying UML specification. The diagrams in this chapter will appear clearer when you finish reading this book and may then serve as a reference for metamodeling (some tools in the market let you make metamodeling and define your proper syntax extension to UML for your own need, not profiles).)

3.1 Core Concepts Compliant to MOF: Modeling Levels

In the real work, we have only objects that interact together to make a working system, no classes. User objects and user data, real or simulated, are found at the "User Object Level" called M0 level. They are images of real objects that we manipulate in everyday life and M0 starts as our reference abstraction level.

As many objects can share common properties, to economize description and make models clearer and manageable, we can factorize properties of similar objects. For instance, the four tires of our car need only a unique description even if all these four objects are distinct and will be worn out differently with time (the four tires have four different states but a same structural description). If there are 20 cereal boxes on the shelf of a store, they may be categorized

Table 3.1.0.1 A four-layer architecture of the UML, which is defined at M2 (specification level or metamodel viewed by users). Users build software at level M1 (user model) to create a running system at M0 (objects and data). The meta-metamodel at M3 level is the basic concept level used for managing interoperability of object standards

Layers viewed by users	*Levels used in this book*	*Scope*	*Defines*	*Constituents*
Meta-metamodel	Basic concept *M3 level*	metamodeling	Language for specifying metamodels	Metaclass, MetaAttribute, MetaOperation
Metamodel	UML specification *M2 level*	UML specification	Language for creating a user model	Class, Attribute, Operation
User model	User model *M1 level*	Project using UML	Language to describe your project	Client, Company, Order_Status, Edit_Order()
User objects and data	User objects *M0 level*	Run-time system	A working system	Order_1234, Client_Paul, Order_Status = "Paid"

into a unique description. The main categorizing mechanism of object technology is *classification*. For the car description, we need to create a *Tire* class with *attributes* to describe tire size, load index, speed rating, etc., and *methods* (operations) to show that tires and wheels must rotate to move car.

The *Tire* class belongs to the User model depicted at M1 level. We need to create at the same time the class *Car* and thousands of classes to describe all other car components. As a car is a complex object, a flat list of thousands of classes is an unstructured set and an unmanageable project. We must reorganize classes into packages, connect classes together via a set of relationships, and describe them with various semantics to account for the way objects inside a car are connected together. So, the user model contains, besides classes, all relationships and artifacts to make models more understandable.

To make user models, we need an object-modeling tool like the UML. CASE toolmakers in the market propose a software to model and develop systems. They adhere to the current standard UML defined and maintained by the OMG. The UML is an object modeling language and is, at the time of this writing, the most comprehensive and object-oriented standard. If our "Car" model is called the "model" (user model at M1 level), then, the UML itself is called a *metamodel* (a model to create model).

Metamodels are developed at M2 level. The UML is defined with graphical guidelines, naming conventions, organizational artifacts like packages, association/relationship connection rules, etc. To build correct and understandable models at M1 level, modelers must fully understand UML conventions and rules defined at M2 level. The UML or the metamodel, once understood, allows us to produce application models at level 1 and instantiate a working system at level 0.

As the UML is known as four-layer model architecture, in fact, it has a fourth level called meta-metamodel or M3 level. This "Basic Concept Level" of the UML comprises a set of low-level modeling concepts and patterns that are in most cases too rudimentary or too abstract to be used directly in modeling software applications. The rationale behind this fourth level is multifold. First, the UML is a language among many other object languages developed and maintained by the OMG. This organization needs a common base for all object languages to fully reuse common object concepts, ease data interchange, and automate model transformations. The Infrastructure of the UML defines base classes that form the foundation not only for the UML but also for other objects standards, for instance, MOF (Meta Object Facility, a common object model metalevel for most object standards promoted by the OMG).

Classes, attributes, operations, and associations at M3 level are sometimes referenced in the literature as MetaClass, MetaAttribute, and MetaOperation. As the prefix "meta" could be confusing as the "meta" applied to model is one "meta" higher than applied to elements they describe (see 1st and 3rd columns

of Table 3.1.0.1), the understanding of the term "meta" in our text is contextual and we let the readers interpret these concepts at their proper level according to the discussion context. The OMG is currently upgrading all of UML chunks to version 2. It is a large specification, made up of four parts:

- **Infrastructure**. This package is reused at several metalevels in various OMG specifications to deal with general modeling process.
- **Superstructure**. This package is used to define specifically UML.
- **Object Constraint Language**. This package defines a formal language used to describe string expressions on UML models. UML diagrams are not refined enough to provide all the relevant aspects of a complete specification. Modelers need to describe constraints and requirements in natural language. However, natural language carries ambiguities incompatible with programming, so a formal language OCL (Object Constraint Language) has been developed. Typically, the OCL is used to specify queries, pre/postconditions, operations, types, etc.
- **Diagram interchange**. The goal of this package is to define a smooth and seamless exchange of documents compliant to the UML standard. For instance, in previous version of UML 1.x, XMI has been used to exchange diagram information. If objects can be easily transported from model to model, all their layout information is lost. Layout information locates modeling objects on a diagram, specify their sizes, etc. This limitation is not due to XMI but comes from the fact that UML metamodel has not yet defined a standard way of doing things. Moreover, to assure the exchange of tools that do not understand model elements, but only lines, text, and graphics, bridging from XMI to Scalable Vector Graphics (SVG, an XML-based format that has been adopted as a W3C recommendation) if necessary.

3.2 Infrastructure: Core Package

Hereafter, we make an overview of the M3 level to show how UML is built. Diagrams are inspired from original and public documentation available at the end of 2005 on the OMG web site (Infrastructure Book). For research purposes, please consult the OMG web site. If there is misinterpretation by us, please take the interpretation on the OMG web site as reference. We try to synthesize the information for learning purposes, apologize for such an eventual incident, and would be grateful that somebody reports the error. We prune off details and focus only on the way (philosophy) the UML has been built. As graphical notations are examined in detail within the UML syntax of Superstructure Book, they are not explained in this chapter. As said before, the reader can bypass this section if necessary and come back to it later if needed.

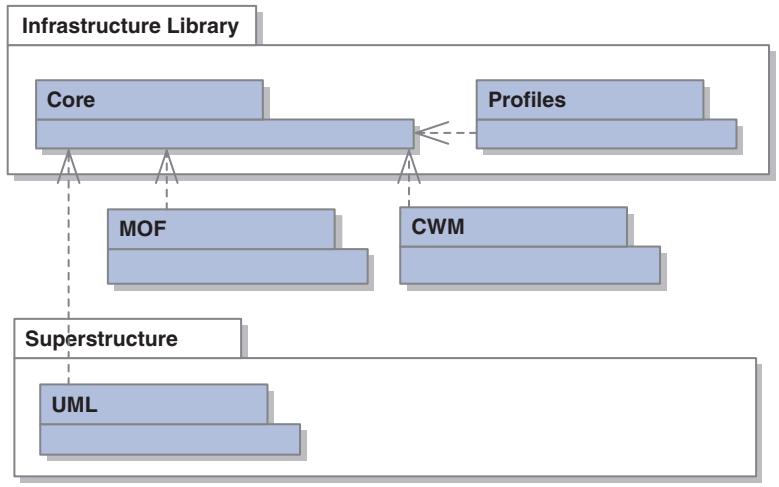

Fig. 3.2.0.1 Reuse of Common Core package by the UML and MOF. The dashed arrow says that the UML is importing and merging modeling elements from Core package. The import and merge notions are detailed in the Superstructure exposé

The Infrastructure of the UML is defined by the *Infrastructure Library* package which is a metalanguage core used in various metamodels, including the MOF, CWM (common warehouse metamodel) and of course, the UML. The MOF is viewed as a language at M3 level, the UML and CWM at M2 level. In Figure 3.2.0.1, the profiles package contains an extension mechanism that allows "second class" customization of UML. "First class" extension could be made via the Infrastructure level. So, it would be interesting for researchers to have an idea of the Infrastructure organization. For the Superstructure, the extension can be made only via profiles, a ruled process defined in UML itself. So, "first class customization" is more powerful but can potentially diverge from UML standard.

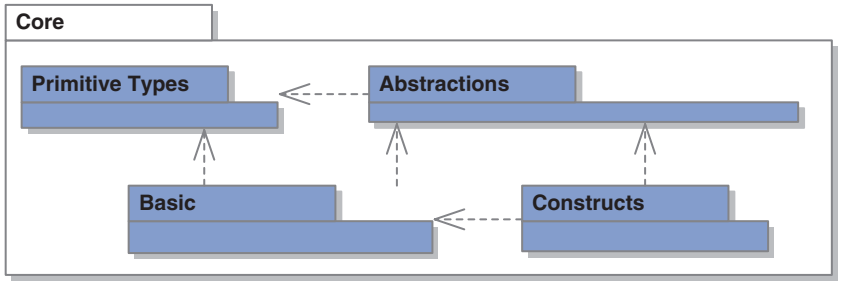

Fig. 3.2.0.2 Four Core packages defined in the Infrastructure. Arrows show their dependencies "merge" or "import" (Fig. 3, Infrastructure Book)

The Core package is divided into four smaller packages to facilitate flexible reuse when creating metamodels. These four subpackages define *metaclasses*, mostly abstract, to be specialized when defining new metamodels. The term *abstract* means that we cannot instantiate any modeling elements directly from an abstract metaclass. Abstract metaclasses are there to support the metamodel framework. A dependency hierarchy exists between those four packages. According to dependency arrows, PrimitiveTypes must be defined first, followed by Abstractions, then Basic, and finally Constructs.

3.2.1 Core::PrimitiveTypes Package

The notation "::" means that PrimitiveTypes package belong the Core package (Fig. 3.2.1.1). PrimitiveTypes package contains four elementary classes stereotyped as <<primitive>>. The signs "<<" and ">>" surrounding any name are used to announce that the named element is a *stereotype* (a preconceived and oversimplified idea of the characteristics which typify a person or thing). These predefined types are commonly used in metamodeling. The <<primitive>> classes are intended for very restrictive graphical uses, specifically to define elementary means to name modeling elements (String class), to count occurrences or multiplicity in a diagram (*Integer*, *Unlimited Natural* classes), and to lay out constraints and evaluate conditions (*Boolean* class). For more advanced context where we need to define ways of writing method specifications, functions, parameters, actions, events, etc., OCL, another important component of

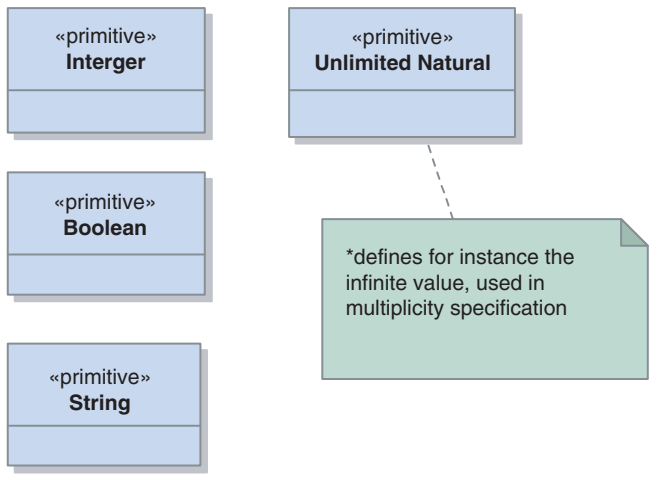

Fig. 3.2.1.1 Core::PrimitiveTypes package contains four classes

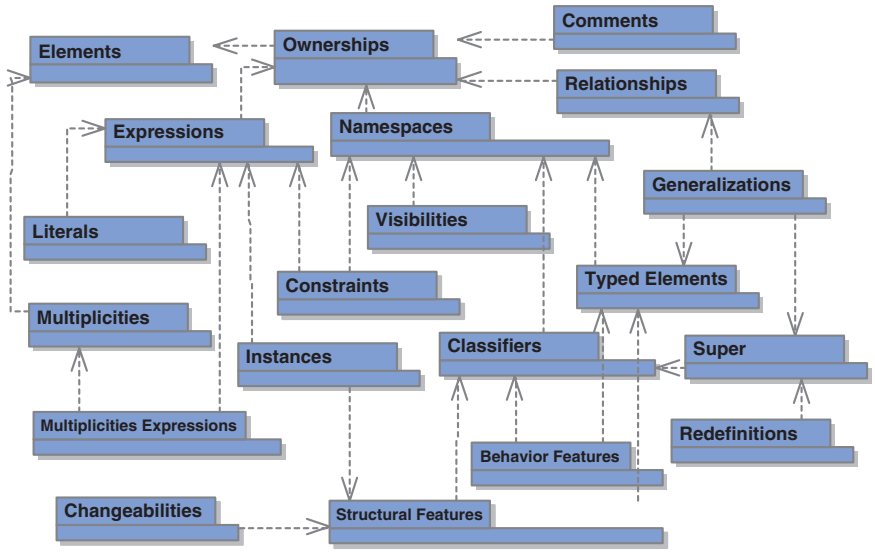

Fig. 3.2.2.1 Contents of Core::Abstractions. This package is composed of 20 elementary packages connected together via a network structure (Fig. 12, Infrastructure Book)

the UML, is more appropriate to unambiguously describe textual constraints in natural language. Please refer to OCL specifications on the OMG site.

3.2.2 Core::Abstractions Package

At the root of Core::Abstractions, we found a unique modeling *Element* which is an abstract metaclass with no superclass. An *Element* is an abstract constituent of a model. The package Core::Abstractions::Elements contains just the class *Element* (Figs 3.2.2.1–3.2.2.3).

An abstract element has the capability of owning other abstract elements. The relationship drawn with a diamond is an aggregation/composition, the diamond is the "owing side" and the side without adornment is the "owned side." The slash ("/ownedElement") is a role specification ("/" means "derived") and, with the association end constraint "{union}," it states that the derived owned Element

Fig. 3.2.2.2 The Core::Abstractions::Elements package contains the metaclass Element (Figs 27 and 28, Infrastructure Book)

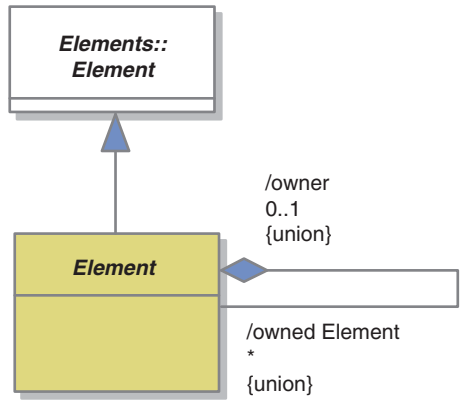

Fig. 3.2.2.3 The Core::Abstractions::Ownerships package expresses the fact that an element can contain other elements (Fig. 50, Infrastructure Book)

association is subset by all composed association ends. The constraints put on an association end are surrounded with curly braces "{ and }" and give a complementary information (constraint) to interpret the role end of an association. "*" or "0..1" are specifications of multiplicities.

> The derived symbol "/" allows marking any model element as derived from other element or elements and thus serving redundant data. The symbol is applicable mainly to attributes and association ends, to indicate that their contents can be computed from other data.

Actually, we cannot find all the information by analyzing only the graphical notation of a diagram. For instance, "an element cannot own itself" or "an element that must be owned must have an owner" are constraints added as textual specifications in the standard. They may also appear through a set of available "queries" accompanying the specification of this package. Together, *constraints* and *queries* must normally complement the graphical notation and specify the package unequivocally. All this information is available in the Infrastructure Book. For this overview, we retain only that a modeling element can own many other elements and the hierarchy "owner/owned" is not limited when applied to the tree structure that this association may generate. This very simple aggregation, together with the defined multiplicity, can in fact generate an infinite tree structure of abstract modeling elements.

A *Comment* is a textual annotation, useful to the modeler that can be attached to an *Element* or a set of Elements. It derives itself from an *Element* class and has an attribute named "body" of type "String" (Fig. 3.2.2.4) that is a primitive type defined in Core::PrimitiveTypes. When deriving from *Ownerships*, any modeling element inherits properties defined in Ownerships and may contain any other elements, in particular, one or several comments. The association at

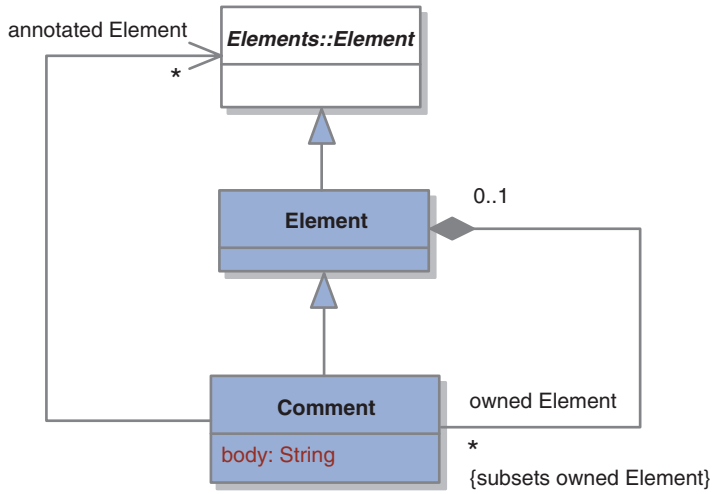

Fig. 3.2.2.4 Core::Abstractions::Comments package (Fig. 20, Infrastructure Book)

the left references the element, which has been commented. The asterisk or star sign (*) means 0, 1, or N elements.

A *Namespace* derives from a *NamedElement* that in turn derives from an *Element* with ownership property. A namespace provides a container for named elements. A *NamedElement* has two attributes. A multiplicity [0..1] applied on an attribute specifies that a *NamedElement* may have no name (which is different from an empty name) or one name. A *NamedElement* has a usual name (that may be ambiguous) and a qualified name (of the form name1 :: name2 :: . . . :: nameN) that allows it to be unambiguously identified within a hierarchy of nested namespaces. It is constructed from the names of the containing namespaces starting at the root of the hierarchy and ending with the name of the *NamedElement* itself. "/" before *qualifiedName* means that it is a derived attribute. The aggregation–composition relationship generates a tree structure. A collection of *NamedElements* are identifiable within the *Namespace*, either by being owned or by being a member, i.e. by importing or inheritance (Fig. 3.2.2.5).

In Figure 3.2.2.6, a *Visibility* provides a mean to constrain the usage of a named element in different namespaces within a model. The diagram shows that a *Visibility* is a *NamedElement* with a *visibility* attribute that takes values from *VisibilityKind* that is of enumeration type. Although enumeration type is defined later in the Core::Basic, *VisibilityKind* is represented temporarily as a stereotype. A *public* element is visible to all elements that can access the contents of the namespace that owns it. A *private* element is only visible inside a namespace that owns it. When a named element ends up with multiple visibilities, *public* visibility overrides *private* visibility.

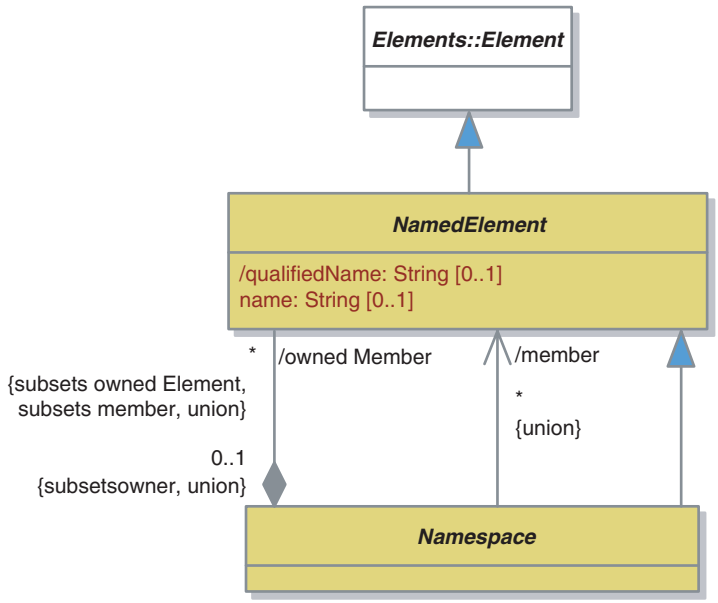

Fig. 3.2.2.5 Core::Abstractions::Namespaces package. A namespace provides a container for named elements (Fig. 48, Infrastructure Book)

Expressions support the specification of values, along with specializations for supporting structured expression trees and opaque expressions (opaque: not interpreted). An *Expression* represents a node in an expression tree. If there are no operands, it represents a terminal node. If there are operands, it represents an operator applied to those operands. Various UML constructs use expressions, which are linguistic formulas, that yields value when evaluated in a context.

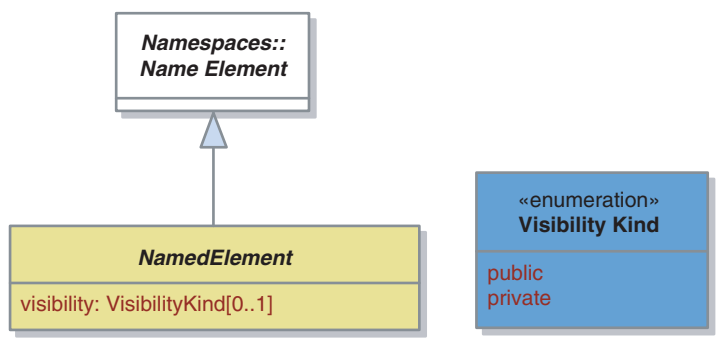

Fig. 3.2.2.6 Core::Abstractions::Visibilities package. This package defines how a element can be visible to other modeling elements (Fig. 62 and 63, Infrastructure Book)

Combined with properties defined in Ownerships package, a complex expression may be generated as a structured tree of symbols. An opaque expression contains language-specific text strings used to describe a value or values, and an optional specification of the corresponding language.

A *Constraint* is a condition or restriction expressed in natural language text attached to the constrained elements. A constraint may or may not have a name; generally it is unnamed. It contains a value specification indicating a condition that must be evaluated to *true* by a correct design. A constraint is defined within a context that is a namespace so a namespace can own constraints. It does not necessarily apply to the namespace itself, but may apply to elements in the namespace. Owned rules are well-formed rules for the constrained elements. To be well formed, an expression must verify the type of conformance rules of the language (e.g. we cannot compare an Integer with a String) (Figs 3.2.2.7–3.3.3.9).

A *Classifier* is defined at its simplest form in a Core::Abstractions package (Fig. 3.2.2.10). It is an abstract namespace whose members can refer to features. A classifier describes a set of instances that have features in common. A *Feature* can be of multiple classifiers.

The Super package (Fig. 3.2.2.11) provides mechanisms for specifying later generalization relationships between classifiers. Any classifier may reference more general classifiers in the reference hierarchy. As a consequence, an instance of a specific classifier is also an (indirect) instance of each of the general classifiers. Therefore, features specified for instances of the general classifier

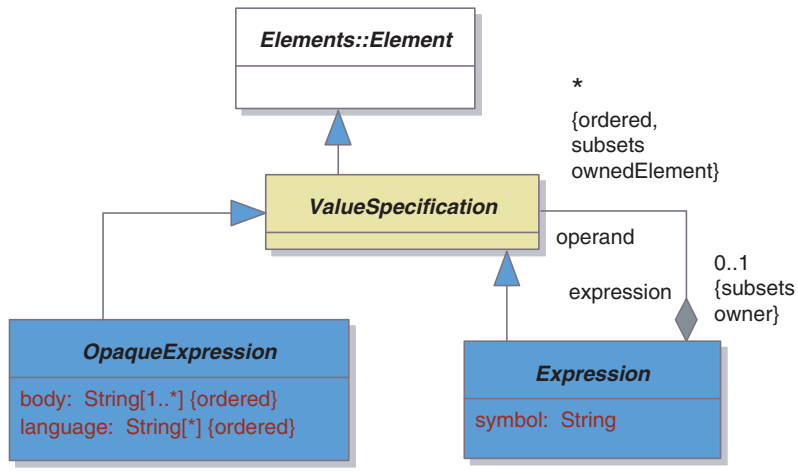

Fig. 3.2.2.7 Core::Abstractions::Expressions package. Expressions are used to support structured expression trees (Fig. 30, Infrastructure Book)

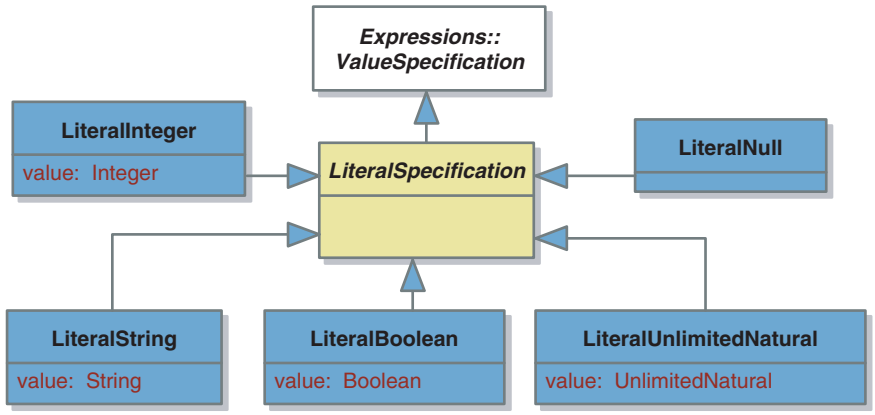

Fig. 3.2.2.8 The Core::Abstractions::Literals package specifies metaclasses for specifying literal values. A literal Boolean is a specification of Boolean value. A literal integer contains an integer-valued attribute, etc. (Fig. 40, Infrastructure Book)

are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

TypedElement and *Type* (Fig. 3.2.2.12) are abstract classes derived from *NamedElement*. The type constrains the range of values represented by a typed element. Simply defined at this stage, a type represents a set of values. According to this definition, a *TypedElement* requires at least a *Type* [0..1] as part of its definition. It does not itself define a *Type*.

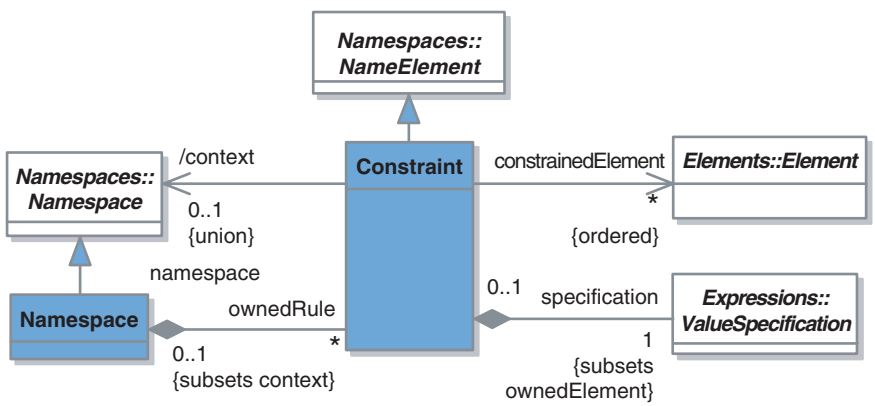

Fig. 3.2.2.9 Core::Abstractions::Constraints package. Constraints use Expressions to textually define conditions that must be satisfied (expressions evaluated to true) in a model (Fig. 23, Infrastructure Book)

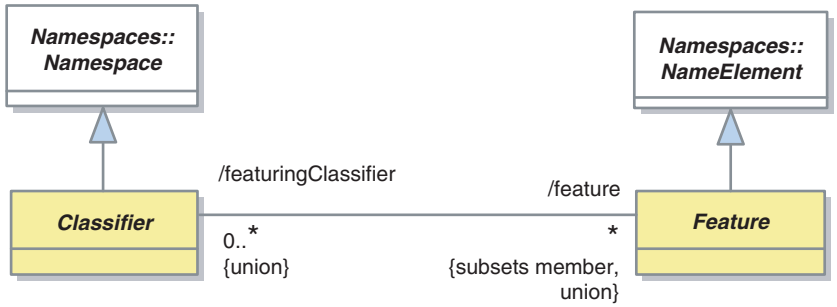

Fig. 3.2.2.10 Core::Abstractions::Classifiers package (Fig. 18, Infrastructure Book). Classifier in the Core package is in its simplest form. It is a namespace and refers to (not contains) features that are namespaces also

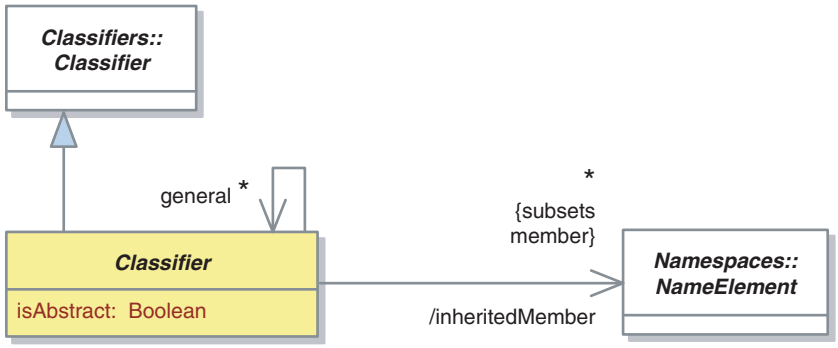

Fig. 3.2.2.11 Core::Abstractions::Super package (Fig. 58, Infrastructure Book)

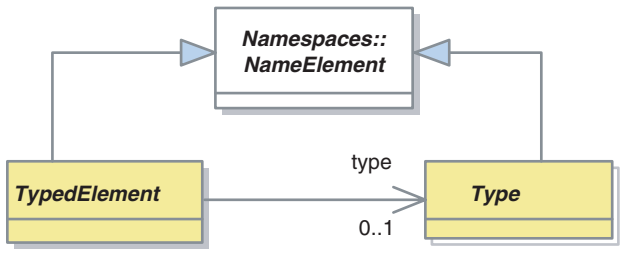

Fig. 3.2.2.12 Core::Abstractions::TypeElements package (Fig. 61, Infrastructure Book)

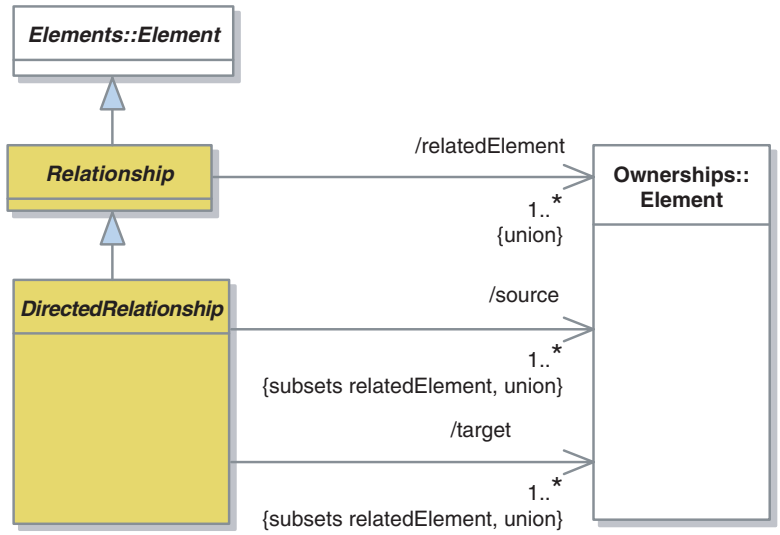

Fig. 3.2.2.13 Core::Abstractions::Relationships package (Fig. 54, Infrastructure Book)

A relationship is an abstract metaclass that references one or more related elements. A *DirectedRelationship* derives from the *Relationship* and references one or more *source* elements and one or more *target* elements (Fig. 3.2.2.13).

A *Generalization* (Fig. 3.2.2.14) is a directed relationship between a specific classifier and a more general classifier. Derived simultaneously from Core::Abstractions::TypedElements::Type and Core::Abstractions:: Super:: Classifier, a classifier in this Generalizations package is a type and can own many generalization relationships.

This package adds the capacity of redefining model elements used in the context of a generalization hierarchy. The statement in the standard is "A redefinable element is an element that, when defined in a context of a classifier, can be redefined more specifically or differently in the context of another classifier that specializes the context classifier" (Fig. 3.2.2.15) The detailed semantics of redefinition varies for each specialization of RedefinableElement.

A *StructuralFeature* (Fig. 3.2.2.16) is a typed feature of a classifier that specifies the structure of instances of the classifier. A *StructuralFeature* is both a *TypeElement* and a *Feature*. The Changeabilities (Fig. 3.2.2.17) package defines when a structural feature may be modified by a client by acting on the *isReadOnly* meta attribute.

An *InstanceSpecification* (Fig. 3.2.2.18) is a model element that represents an instance of a modeled system. It can take various kinds (object if instanced from class, link if instanced from relationship, collaboration, etc.). An instance specification reveals the existence of an entity in the modeled system, and

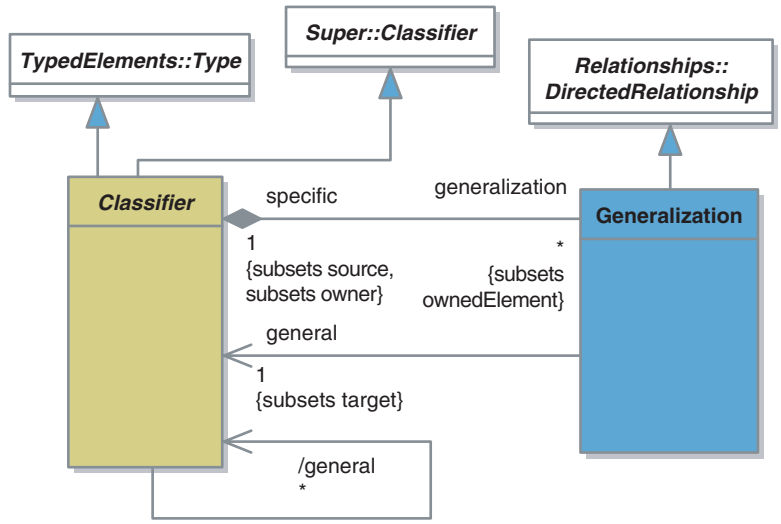

Fig. 3.2.2.14 Core::Abstractions::Generalizations package (Fig. 32, Infrastructure Book)

completely or partially describes this entity. This description embraces: classification of the entity with one or more classifiers, precision on the kind of instance, and specification of values for defining structural features. Details can be incomplete since the purpose of an instance specification is to show what is of interest about an entity in a modeled system. From this metalevel diagram, *InstanceSpecification* is a concrete class derived from *NamedElement*. It references one or many classifiers, has possibly *ValueSpecification* to build the instance, and has many slots, each described by a *StructuralFeature*. The

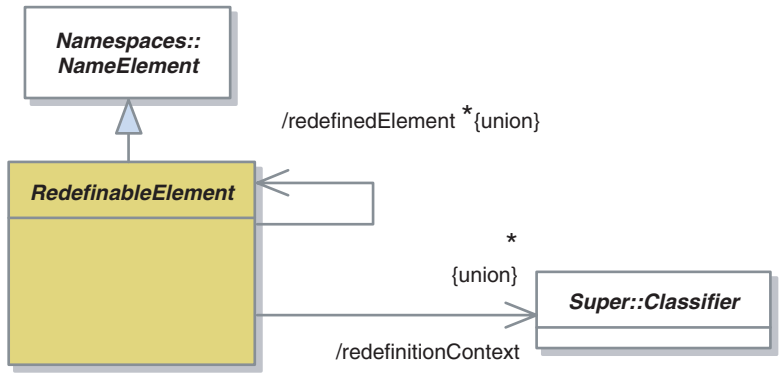

Fig. 3.2.2.15 Core::Abstractions::Redefinitions package (Fig. 52, Infrastructure Book)

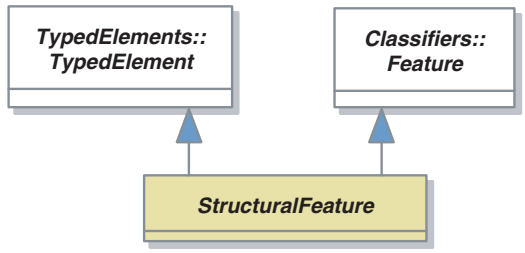

Fig. 3.2.2.16 Core::Abstractions::StructuralFeatures package (Fig. 56, Infrastructure Book)

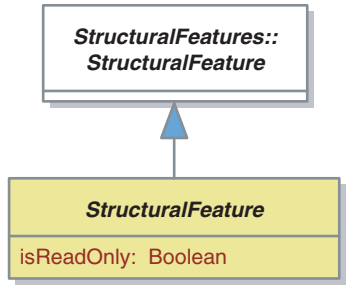

Fig. 3.2.2.17 Core::Abstractions::Changeabilities package (Fig. 16, Infrastructure Book)

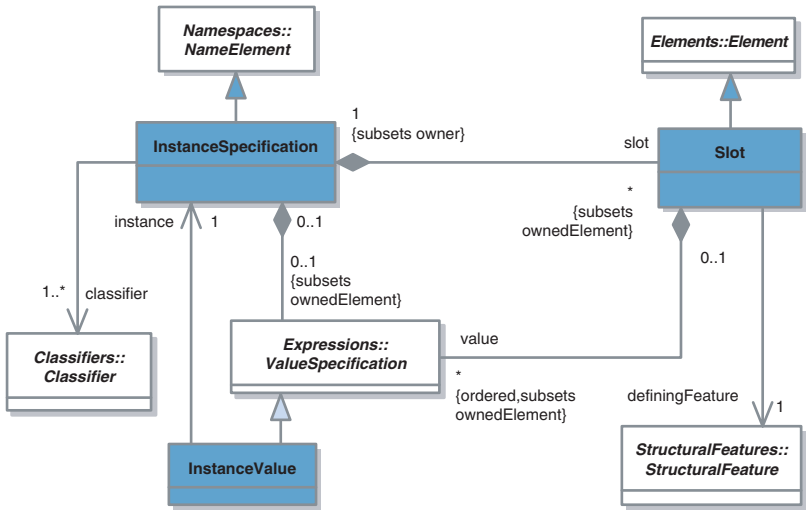

Fig. 3.2.2.18 Core::Abstractions::Instances package (Fig. 35, Infrastructure Book)

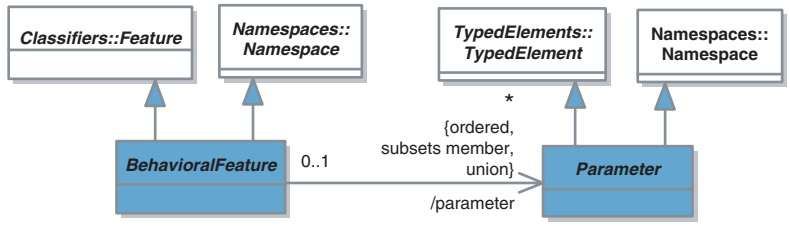

Fig. 3.2.2.19 Core::Abstractions::BehavioralFeatures package (Fig. 14, Infrastructure Book)

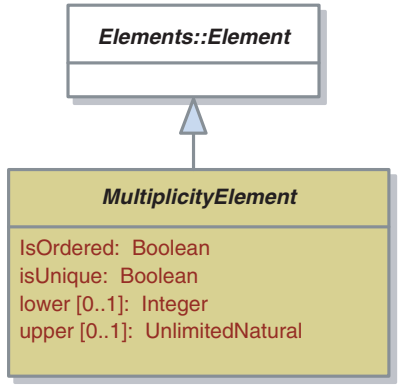

Fig. 3.2.2.20 Core::Abstractions::Multiplicities package (Fig. 42, Infrastructure Book)

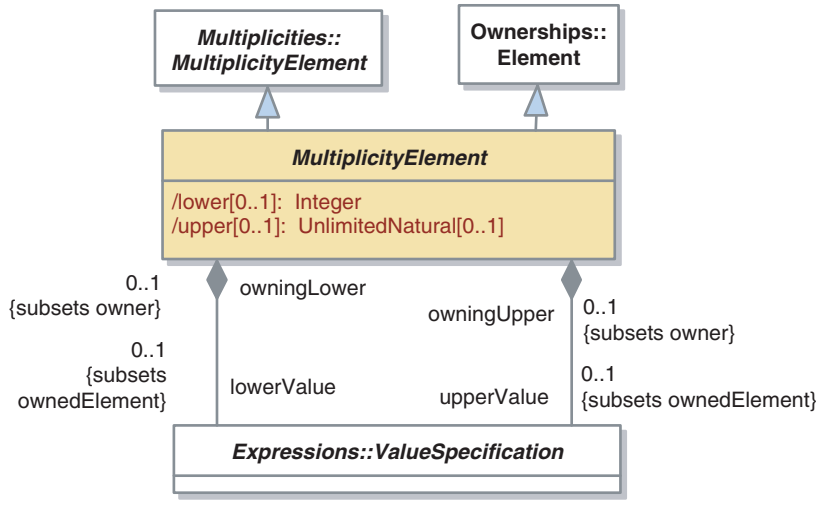

Fig. 3.2.2.21 Core::Abstractions::MultiplicityExpressions package (Fig. 46, Infrastructure Book)

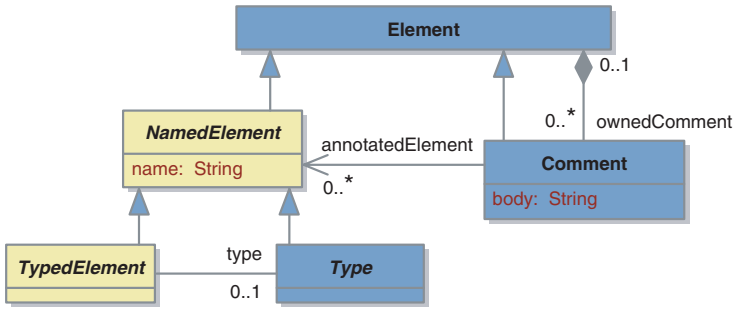

Fig. 3.2.3.1 Core::Basic::Types package (Fig. 65, Infrastructure Book)

structural features of classifiers like *attributes, link ends, parts*, etc. in UML are called *Slots*. A value in a Slot is a *ValueSpecification*.

A *BehavioralFeature* (Fig. 3.2.2.19) is a feature of a classifier that specifies an aspect of behavior of its instances. A *Parameter* is a specification of an argument used to pass information into or out of an invocation of a behavioral feature. *Parameter* has a type since it is derived from *TypeElement* and *Namespace*.

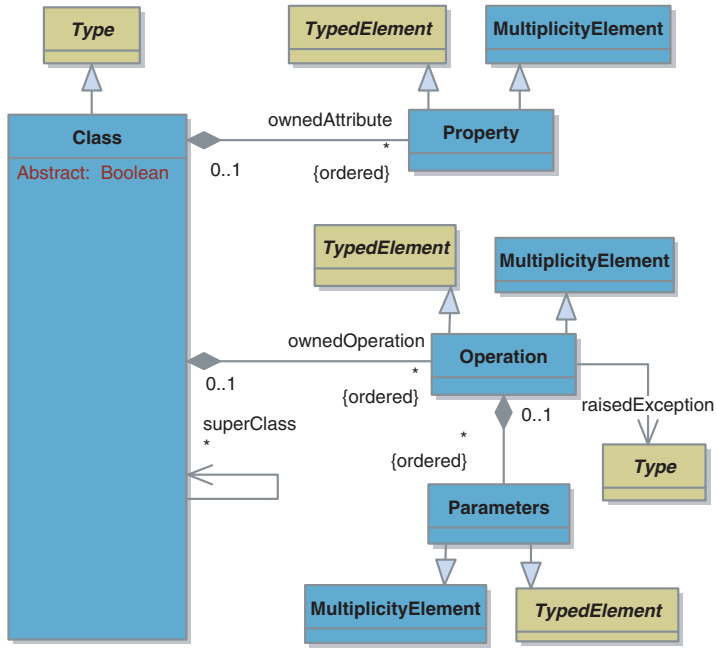

Fig. 3.2.3.2 Core::Basic::Classes package (reduced version of Fig. 66, Infrastructure Book)

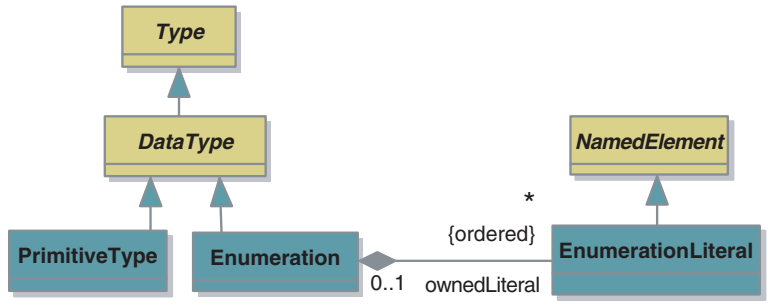

Fig. 3.2.3.3 Data types in the Core::Basic::DataTypes package (Fig. 67, Infrastructure Book)

A *Multiplicity* (Fig. 3.2.2.20) is an inclusive interval of nonnegative integers beginning with a *lower* bound and ending with an *upper* bound, which can be infinite. Multiplicities are mostly used to specify later association ends. A multiplicity specifies the allowable cardinalities of an element and regulates the number of instances we can generate from this element.

This MultiplicityExpressions package extends the multiplicity capabilities to support the use of expressions. *MultiplicityElement* is an abstract metaclass, which includes optional attributes for defining the bounds.

3.2.3 Core::Basic Package

Core::Basic package provides a minimal class-based modeling language on top of which more complex languages can be built. Core::Basic was effectively reused in MOF and contains four diagrams: Types, Classes, DataTypes, and Packages. Core::Basic imports model elements from PrimitiveTypes package and contains metaclasses derived from Core:: Abstractions.

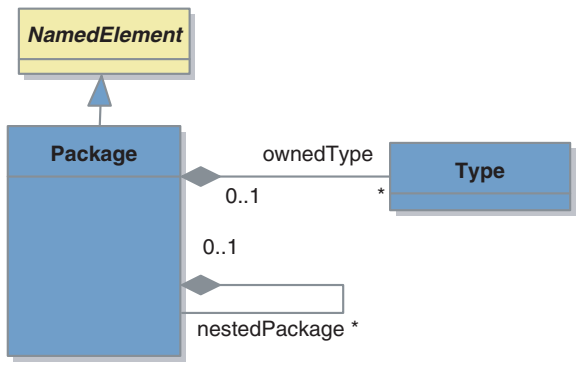

Fig. 3.2.3.4 Core::Basic::Packages package (Fig. 68, Infrastructure Book)

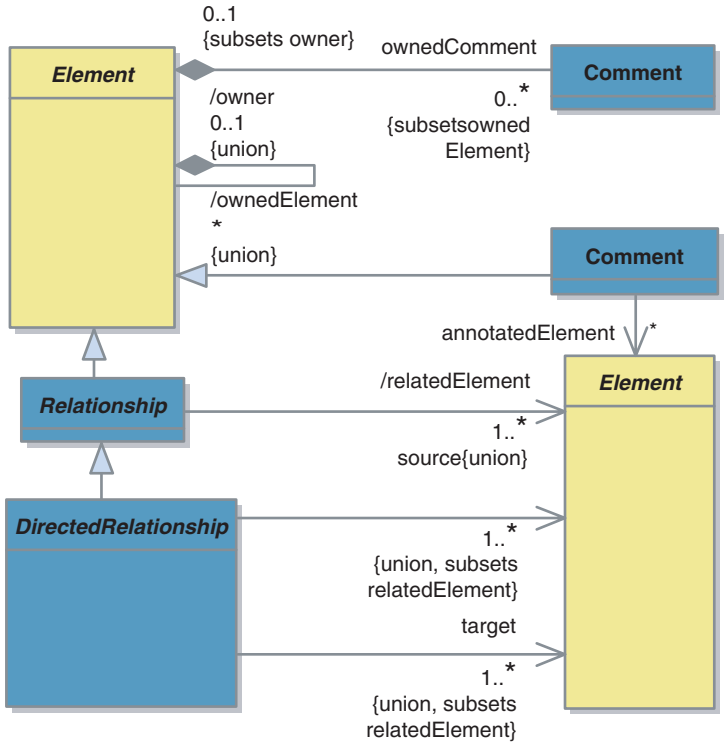

Fig. 3.2.4.1 Core::Constructs::Root package (Fig. 71, Infrastructure Book)

Figure 3.2.3.1 defines abstract metaclasses that deal with naming and typing of elements. *Element* is the root element defined in Core::Abstractions. *NamedElement* represents "element with a name." A *Type* is a named element used to specify a type. *TypedElement* is then a named element with a defined type. Each element may have zero or several comments attached to it.

Next, the Core::Basic::Classes package was defined. In Figure 3.2.3.2, a *Class* that has later objects as instances is a typed concrete metaclass in this Core::Basic package. A class participates in inheritance hierarchy and has properties and operations. The self loop says that a class can be a superclass of an infinite number of classes. Multiple inheritance is allowed.

Conceptually, in this Core::Basic::Classes package, there is no difference between an attribute and an association end besides their different notations. If a property is owned by a class, it is an attribute. If a property is owned by an association, it represents an end of the association. They are all "properties." The type of the operation, if any, is the type of the result returned by the operation. The multiplicity of the operation is the multiplicity of its result. An operation

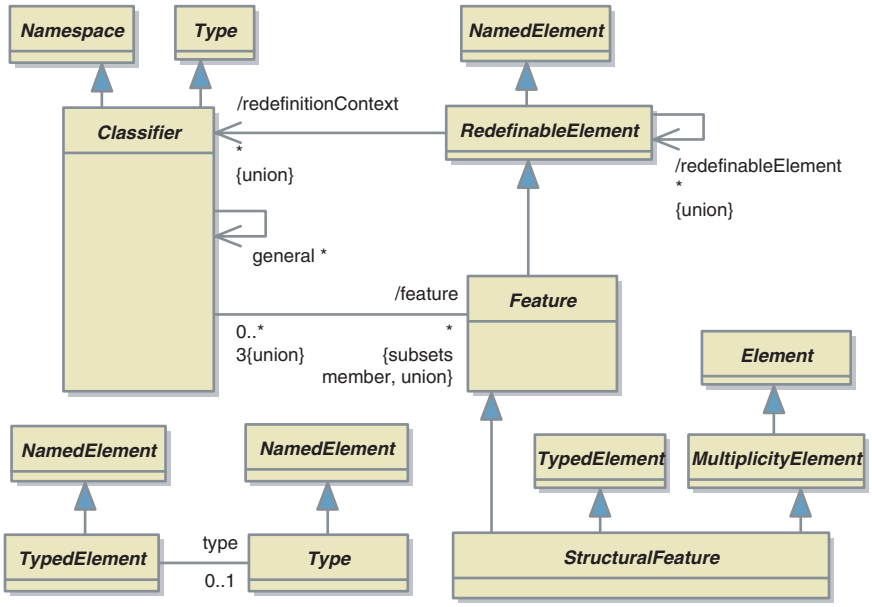

Fig. 3.2.4.2 Core::Constructs::Classifiers package (Fig. 84, Infrastructure Book)

can be associated with a set of types that represent exceptions raised by the operation. Each parameter of the operation also has a type and multiplicity.

The next package is the Core::Basic::Datatypes. In Figure 3.2.3.3, *DataType* is an abstract class that acts as a common superclass for different kind of data types. *PrimitiveType* is a data type. Primitive types used at the Core level are *Integer, Boolean, String*, and *UnlimitedNatural*. An *Enumeration* is composed of a set of literals used as its values.

The last package is Core::Basic::Packages (Fig. 3.2.3.4). Packages provide a way of grouping types and packages for managing a model. In this figure, a package cannot contain itself; but it can include other packages, their contents, and various types. This definition of *Package* allows us to put practically everything in a package, named or unnamed.

3.2.4 Core::Constructs Package

The last package of the Core level is the Constructs package that is dependent on all other packages of the Core.

It merges practically all constructs of other packages. It imports model elements from Core::PrimitiveTypes, it also contains metaclasses from Core::Basic and shared metaclasses from Core::Abstractions obtained by copy. It defines nine new packages: *Root, Expressions, Classes, Classifiers, Constraints, DataTypes, Namespaces, Operations*, *and Packages*. Except

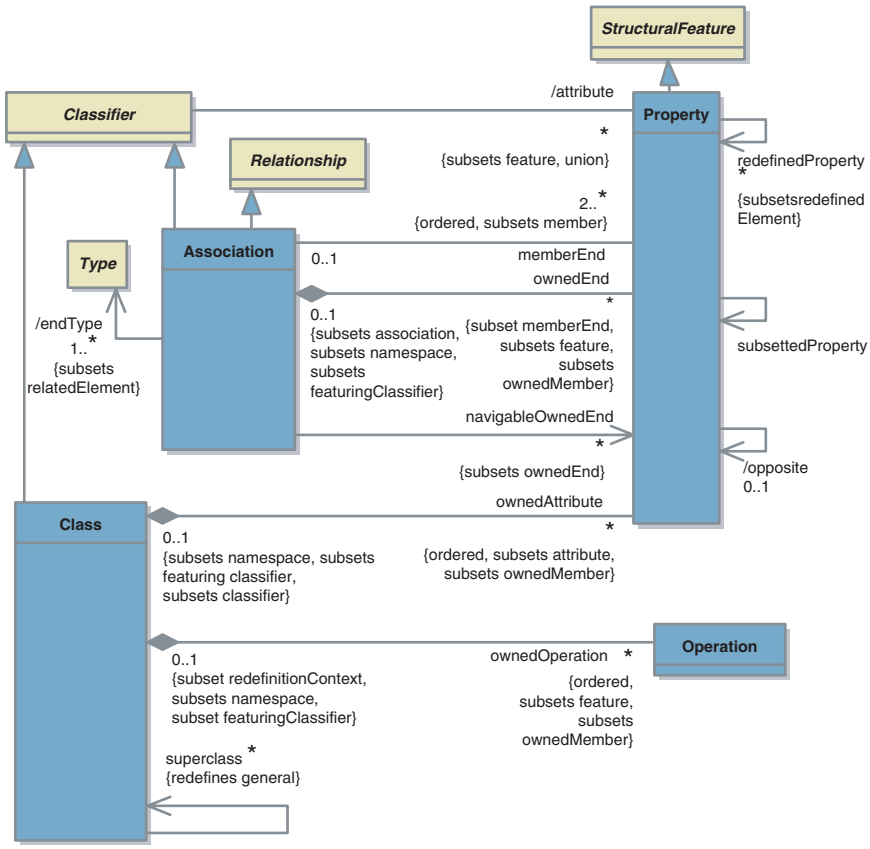

Fig. 3.2.4.3 Core::Constructs::Classes package (Fig. 73, Infrastructure Book)

Operations package, all other names are already present in more elementary packages like Core::Abstractions or Core::Basic. In fact, Classes coming from Core::Basic::Classes or Core::Constructs::Classes are viewed in this namespace organization as two independent packages. We can say that the Core package has two definitions of "Classes." A modeling language specification at M2 level will then have the choice of reusing Classes from Core::Basic or from Core::Constructs from level M3.

The Root diagram of Figure 3.2.4.1 specifies the *Element*, *Relationships*, *DirectedRelationship*, and *Comment* constructs. All these concepts are already defined in Core::Abstractions; they are simply merged together inside one "Root" package.

This package specifies the concepts *Classifier, TypedElement, Multiplicity Element, RedefinableElement, Feature*, and *StructuralFeature*, merging heavily concepts from Core::Abstractions and Core::Basic.

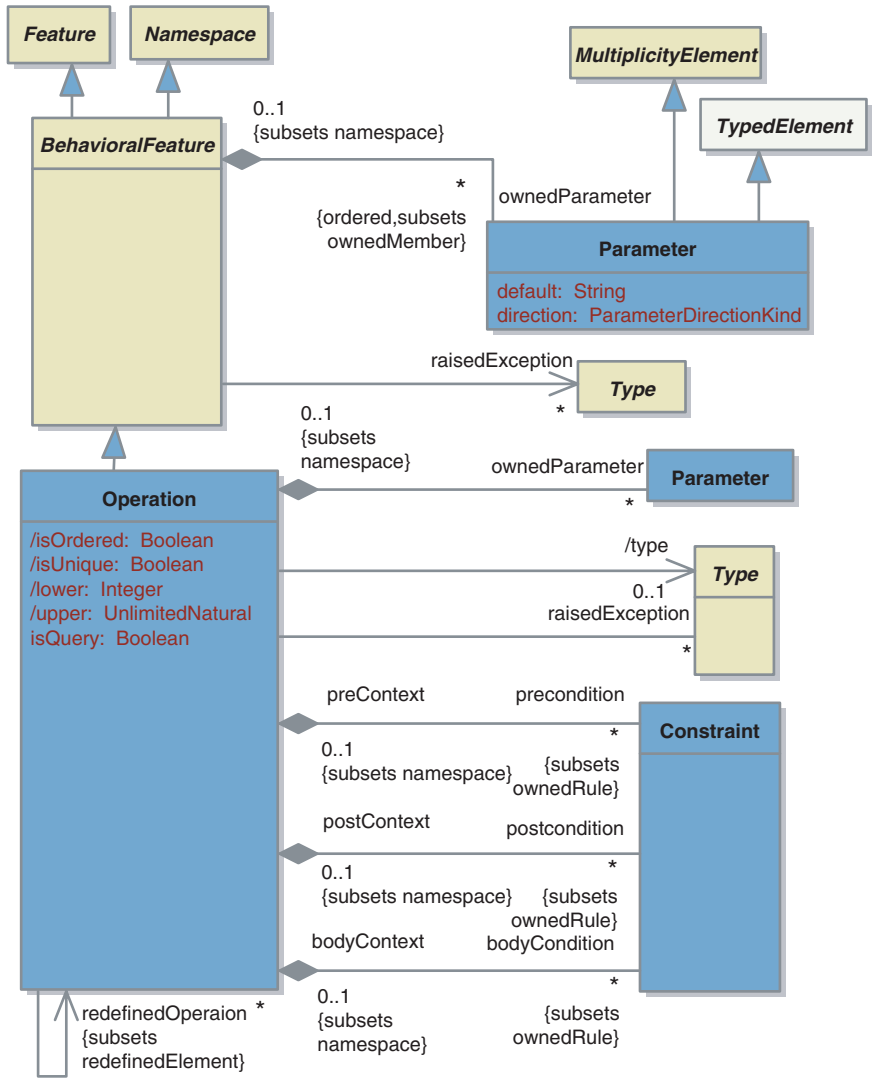

Fig. 3.2.4.4 Core::Constructs::Operations package (Fig. 93, Infrastructure Book)

In the Figure 3.2.4.3, Core::Constructs::Classes package is defined. From this Core::Constructs::Classes package, we learn that a class derives directly from abstract *Classifier*. More precise than the definition of class in Core::Basic::Classes package, a new *Association* class is now derived simultaneously from the Core::Abstract::Classifiers and Core::Abstract:: Relationships packages, so "Property" is now split into two distinct entities, *attribute* and *association end*, through the creation of an *Association* class. Classes have properties; operations and are now connected through associations.

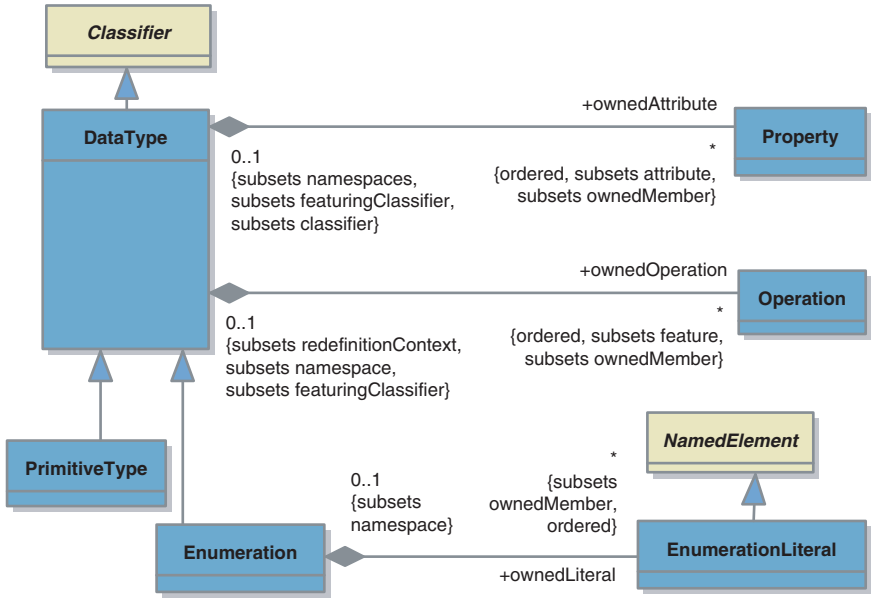

Fig. 3.2.4.5 Core::Constructs::Datatypes package (Fig. 86, Infrastructure Book)

In Figure 3.2.4.4, the Core::Constructs::Operations package is built to specify the *BehavioralFeature, Operation*, and *Parameter* constructs. An operation is a behavioral feature of a classifier and has a set of parameters. *Parameter DirectionKind* is an enumeration type having four values (in, out, inout, return). A parameter specifies how arguments are passed into or out of an invocation. An operation is invoked on an instance of the classifier for which the operation is a feature. The preconditions must be true when the operation is invoked. The postconditions define conditions that must be true when the operation is completed successfully. The body condition constrains the result returned by the operation. An operation may raise an exception during its execution. When an exception is raised, it should not be assumed that the postconditions or body conditions are satisfied. An operation may be redefined in a specialization of the classifier. In this case, it can refine the specification of the operation.

In Figure 3.2.4.5, Core::Constructs::Datatypes package complements the Core::Basic:: Datatypes package by adding a data type to Classifier. A data type in the package Core::Constructs contains attributes to support the modeling of structured data types. There exists a similitude in the way DataType and Class are defined at this stage. Both Class and DataType are derived from Classifier. DataType differs from Class in that instances of DataType are identified only by their values. All copies of an instance of a data type and any instance

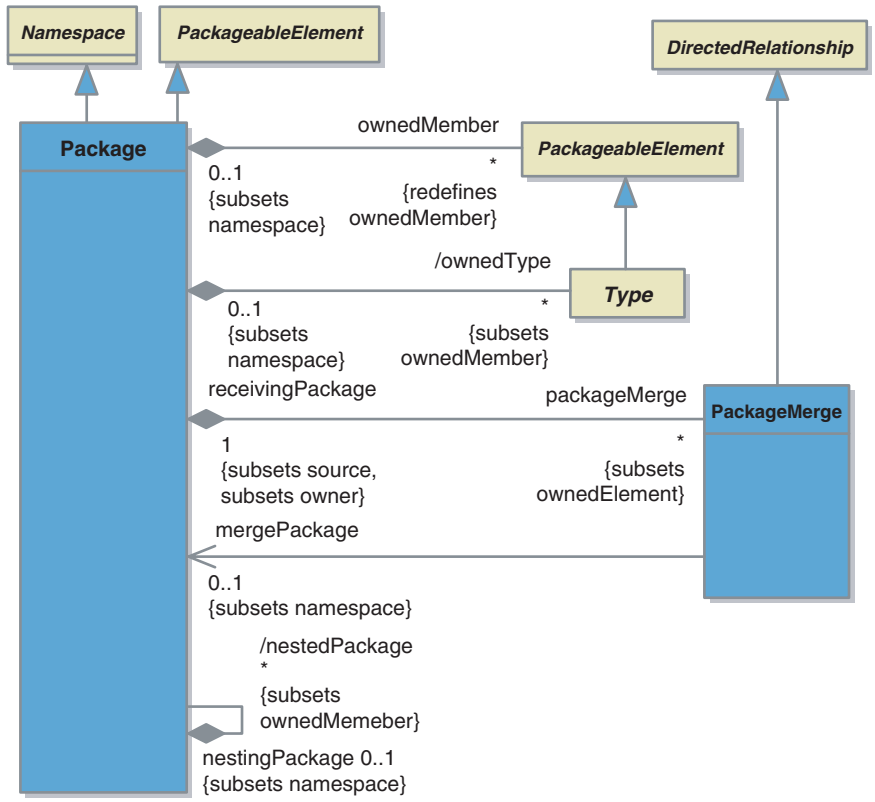

Fig. 3.2.4.6 Core::Constructs::Packages package (Fig. 94, Infrastructure Book)

of that data type with the same value are considered to be the same instance. They are value based.

Core::Constructs::Packages package of Figure 3.2.4.6 specifies a new *Package* and *PackageMerge* constructs. A package is a namespace for its members and may contain other packages. Only packageable elements can be owned members of a package. Being a namespace, a package can import either individual members of other packages or all their members. The principal mechanism governing the construction of package is package merge.

The Core::Constructs::Namespaces package (Fig. 3.2.4.7) specifies the general capacity for any namespace to import all or individual members of packages. An *ElementImport* is a *DirectedRelationship* between an importing namespace and a packageable element or its alias to be added to the namespace of the importing namespace. A *PackageImport* is a *DirectedRelationship* that identifies a *Package* whose members are to be imported by a namespace. The notion of package import differs from the package merge and will be explained later.

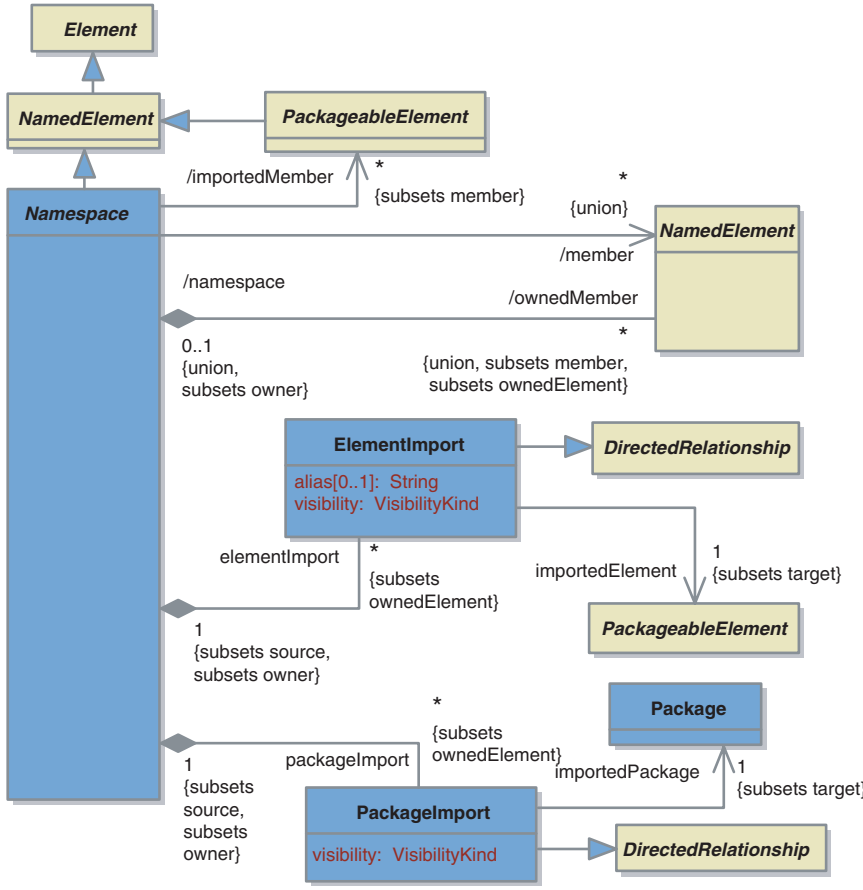

Fig. 3.2.4.7 Core::Constructs::Namespaces package (Fig. 89, Infrastructure Book)

The Core::Constructs::Expressions package is very close to the Core:: Abstractions::Expressions. Instead of the Core::Abstractions::Ownerships:: Element, we now derive an element from *PackageableElement* (named element that may be owned directly by a package) and a *TypedElement* (please consult Core::Abstractions::Expressions for comparison).

The Core::Constructs::Constraints package differs only from Core:: Abstractions::Constraints by the fact that Constraint derives now from a *PackageableElement*, and not from a *NamedElement*.

3.2.5 Core::Profiles Package

In the Core::Constructs::Profiles package, *Profile* is a package issued by itself from a namespace. A *ProfileApplication* indicates which *Profile* has been

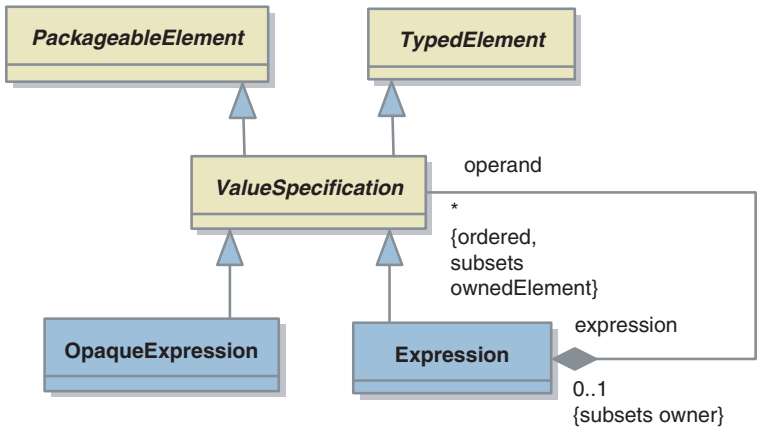

Fig. 3.2.4.8 Core::Constructs::Expressions package (Fig. 72, Infrastructure Book)

applied to the package. It comes from *PackageImport* which is a *DirectedRelationship*. A stereotype defines how an existing metaclass can be extended. A class may be extended by one or more stereotypes. *Extension* is a kind of *Association*. An *ExtensionEnd* is used to tie an extension to a stereotype when extending a metaclass. A stereotype can change the graphical appearance of the extended model by using an attached icon. Finally, metaclass can be individually extended, and so can the complete package.

The Profiles package provides mechanisms for adding new semantics to a metamodel without creating contradictions with existing packages (*profiling* versus *metamodeling*). Rules could be perceived as constraining but the result is worth the exercise. Moreover, Profiles can extend by restricting. For instance, generalization of classes should be able to be restricted to single inheritance (multiple inheritance not allowed), without explicitly assigning any stereotype.

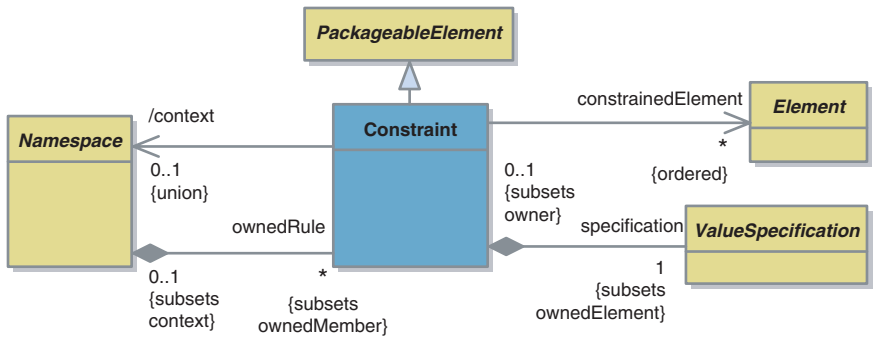

Fig. 3.2.4.9 Core::Constructs::Constraints package (Fig. 85, Infrastructure Book)

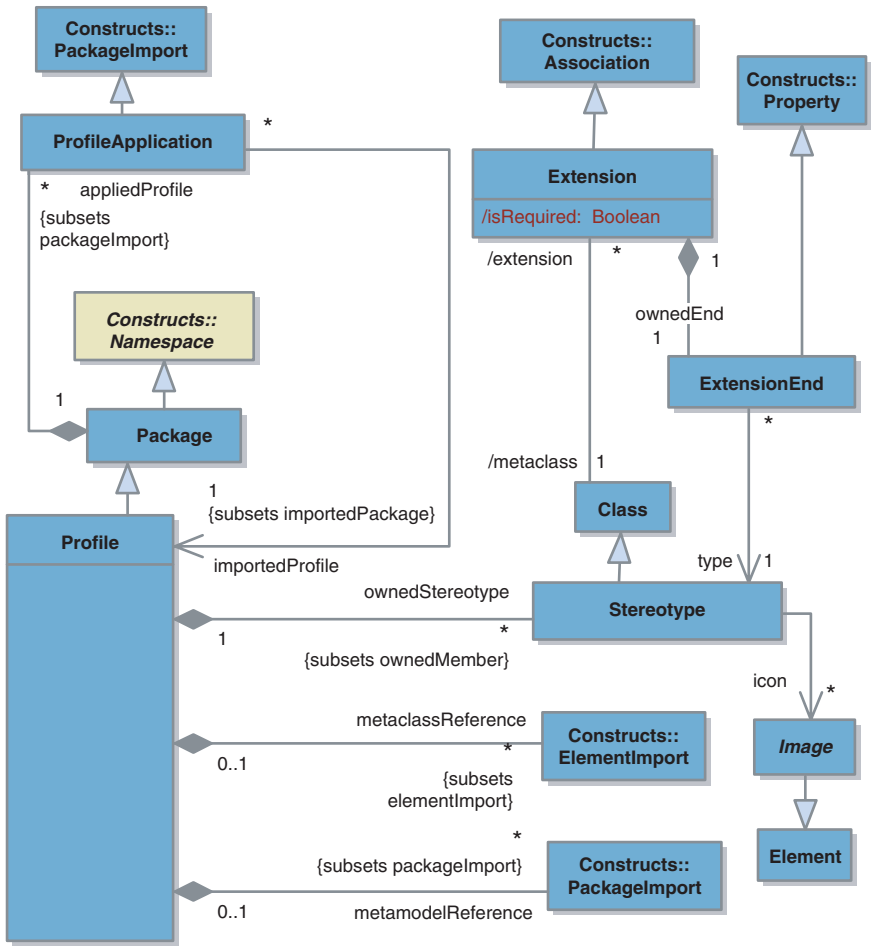

Fig. 3.2.5.1 Core::Profiles package (Fig. 109, Infrastructure Book)

For instance, Java classes do not support multiple inheritance. We do not have to put a stereotype sign <<Java>> on every Java class instances from the metamodel because we restrict Java to single inheritance.

The reference metamodel (in our case UML) is then considered as a "read-only" model that can be extended by profiles without changing the reference metamodel. Such restrictions do not apply to the MOF as it is a Basic Concept model and, from this viewpoint, can be reworked in any direction. People talk of "first class extension" that offer more flexibility in regard to semantics. If we want to define new metamodel, the right way would be from the MOF, not through Profiles. The intentions of Profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific

to a particular domain, platform, or method (second class extension). In other words, the profiling approach does not compromise the existing metamodel and shields user extensions from each other.

From a research point of view, the first class extension mechanism is more interesting to extend the object paradigm towards unexpected frontiers. The Superstructure is tied to the current version of the UML. The Infrastructure of the UML, just exposed, is a good example of that could be inspired as part of ontology engineering, future object concept building.