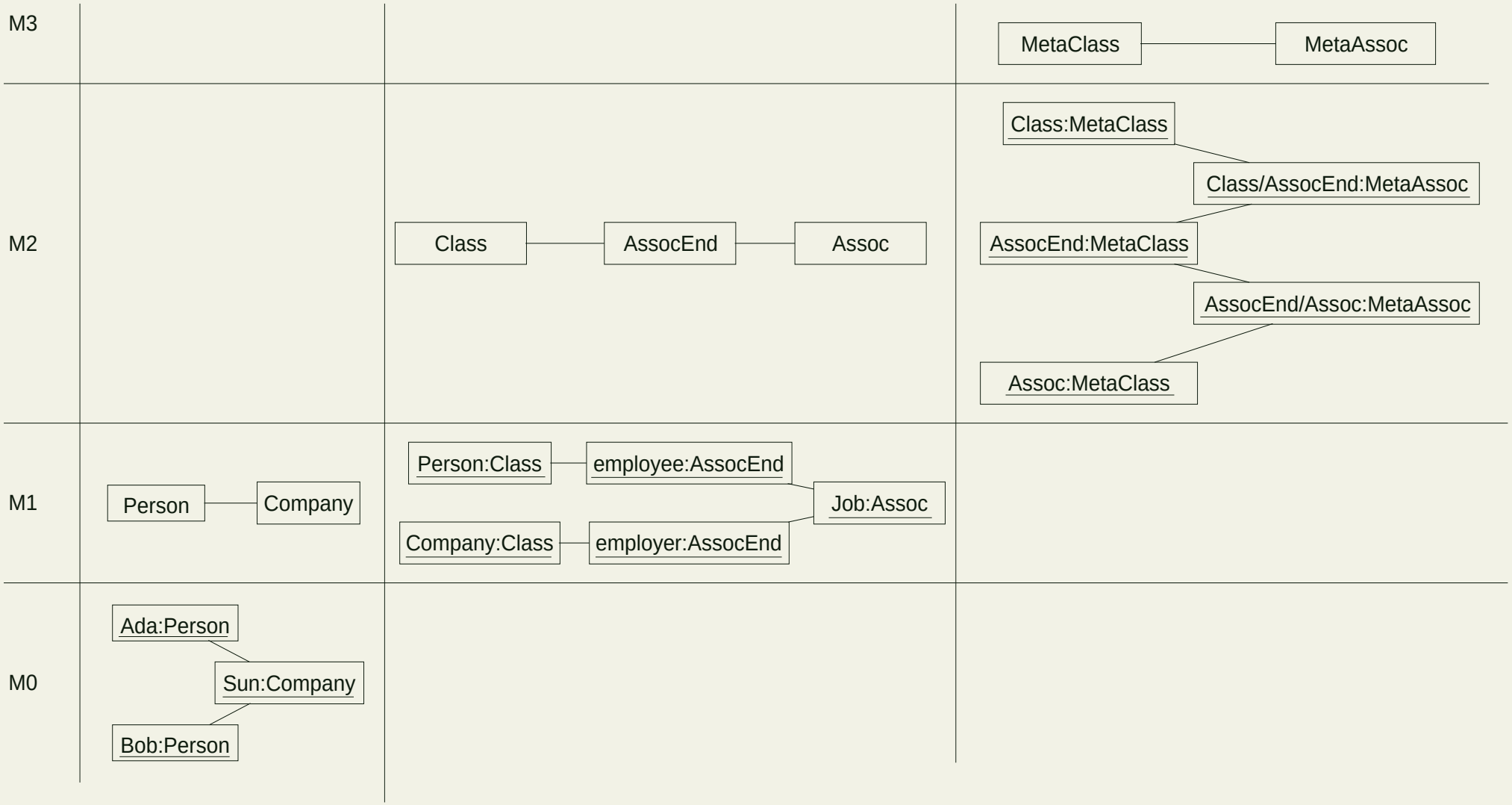
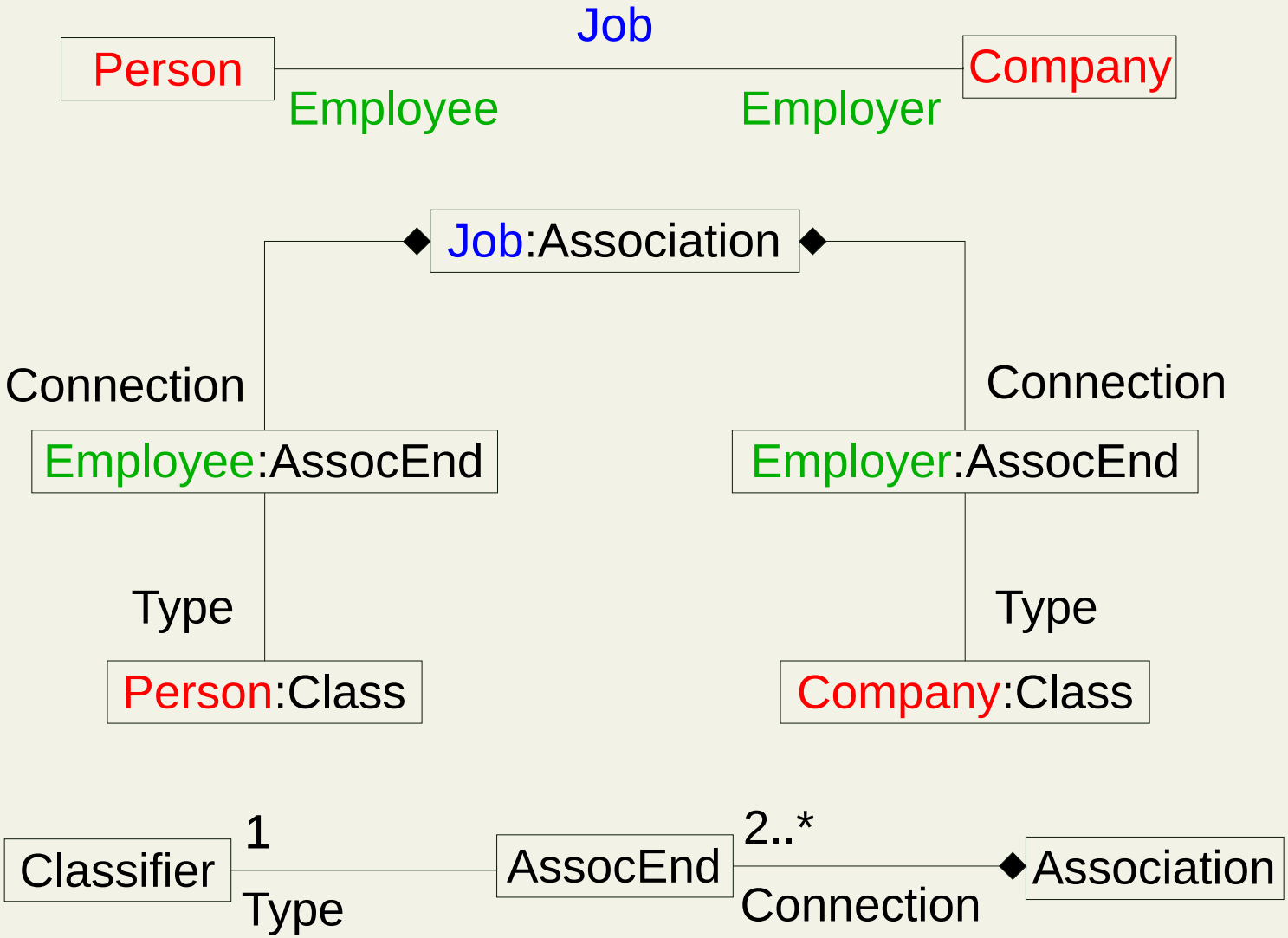
8. UML Metamodel

8.1 Getting Started

Table 2-1 from Specification

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*


M3
MetaClass
MetaAssoc
M2
Class:MetaClass
Class/AssocEnd:MetaAssoc
Class
AssocEnd
Assoc
AssocEnd:MetaClass
AssocEnd/Assoc:MetaAssoc
Assoc:MetaClass
M1
Person:Class
employee:AssocEnd
Job:Assoc
Person
Company
Company:Class
employer:AssocEnd
M0
Ada:Person
Sun:Company
Bob:Person


Job
Person
Company
Employee
Employer
Job:Association
Connection
Connection
Employee:AssocEnd
Employer:AssocEnd
Type
Type
Person:Class
Company:Class
Classifier
1
Type
AssocEnd
2..*
Connection
Association

8.2 Class Diagrams for UML Core

Figure 2-1 Top-Level Packages

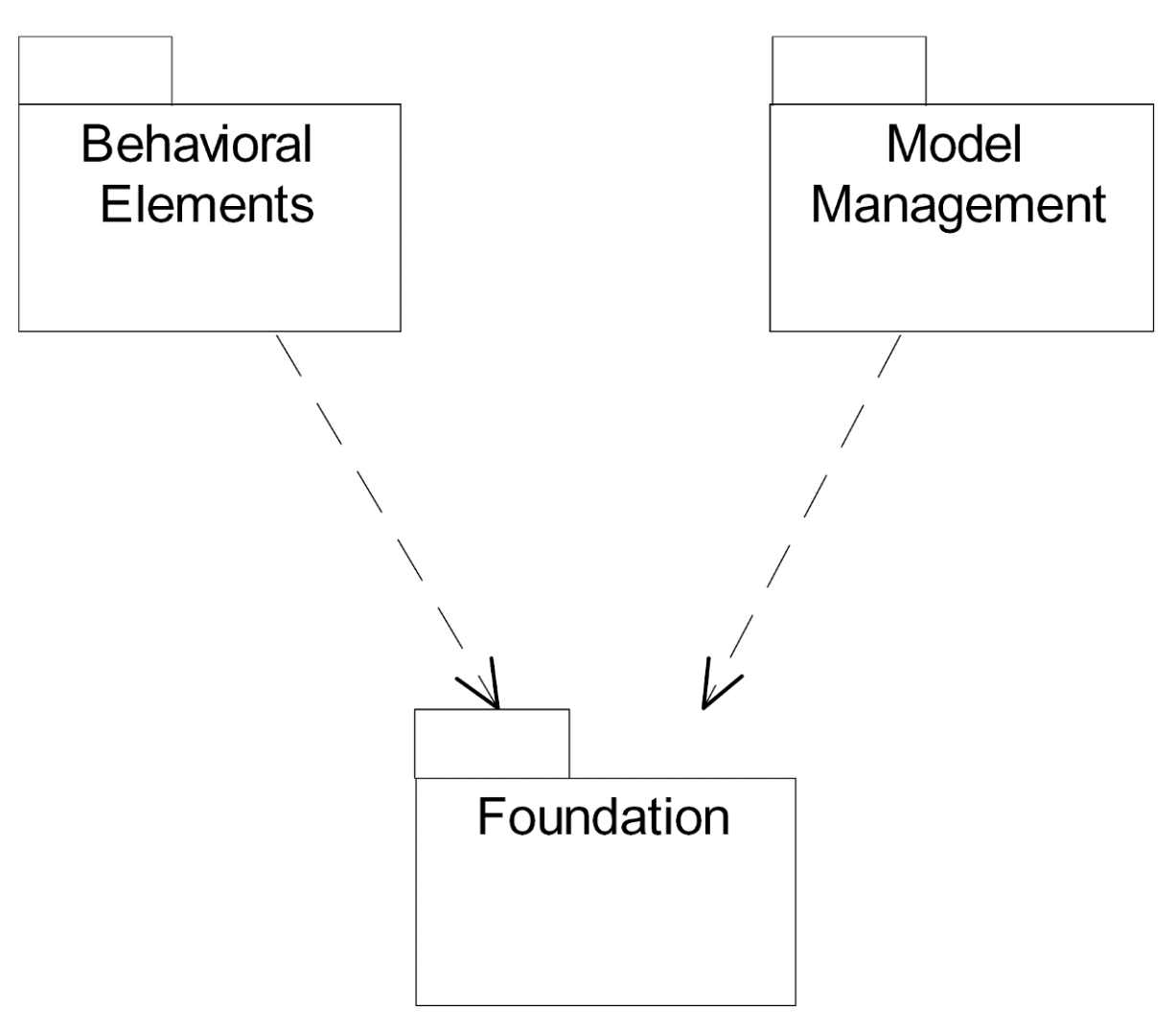


Figure 2-2 Foundation Packages

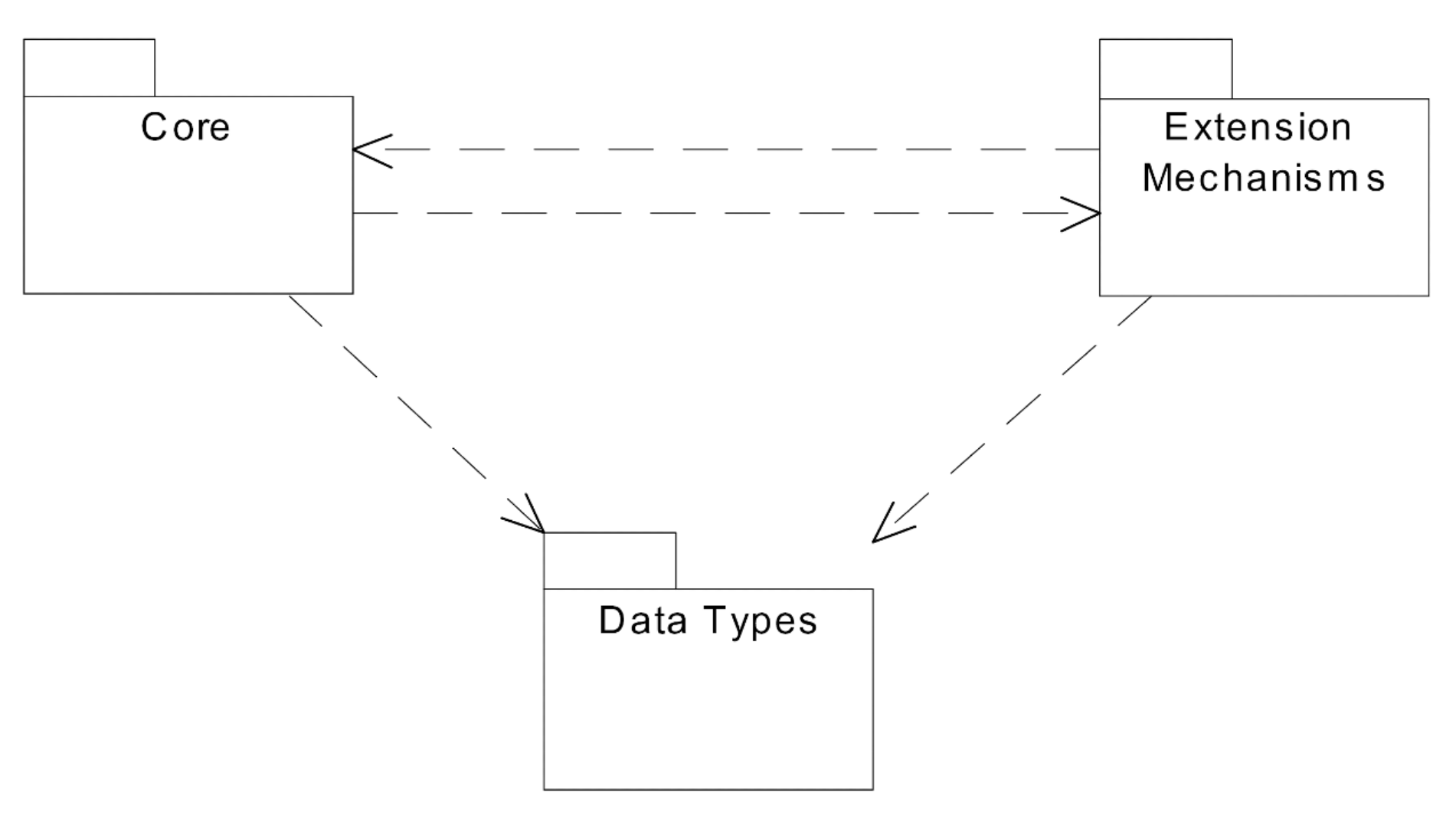


Figure 2-3 Behavioral Elements Packages

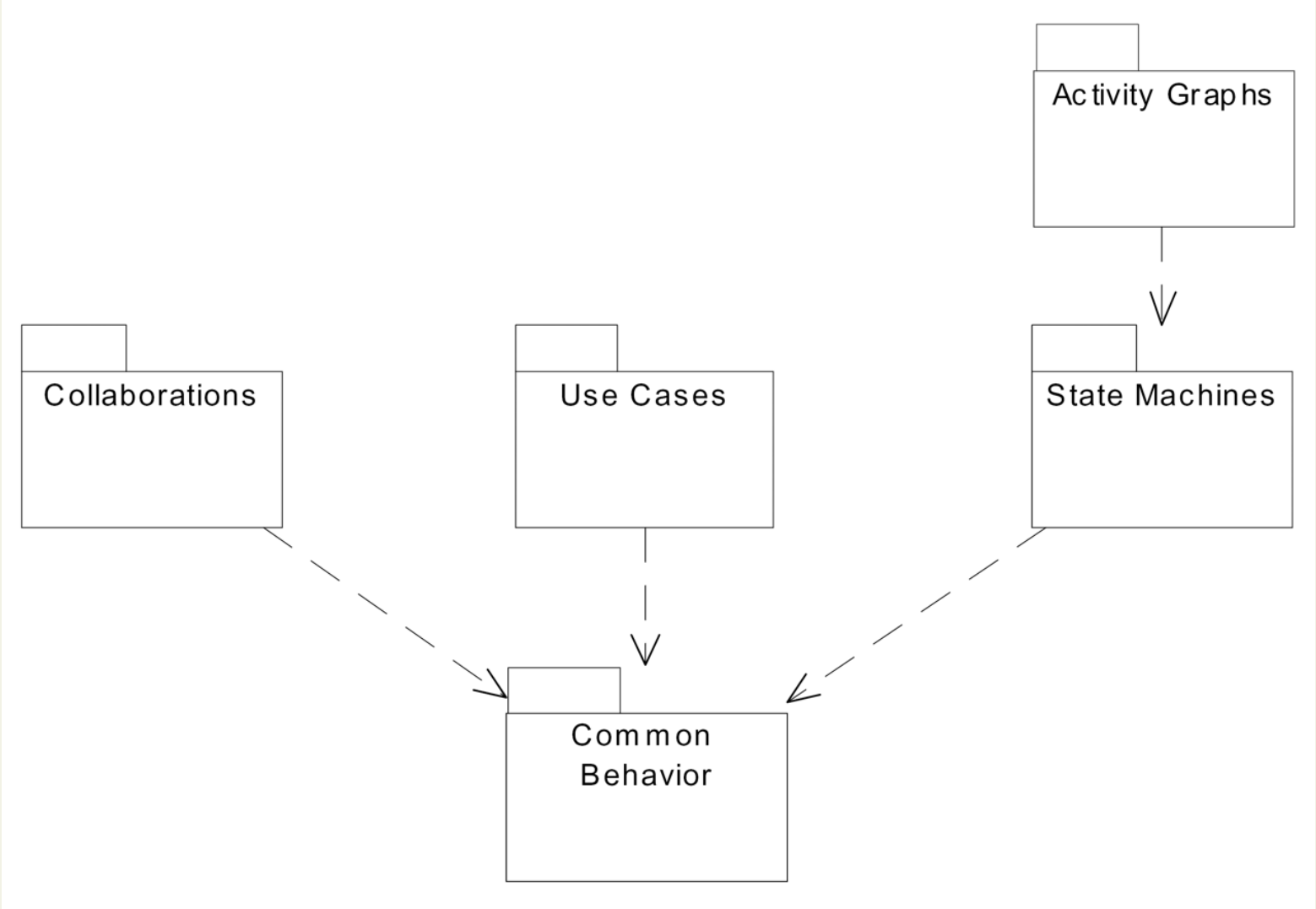


Figure 2-4 Foundation Packages

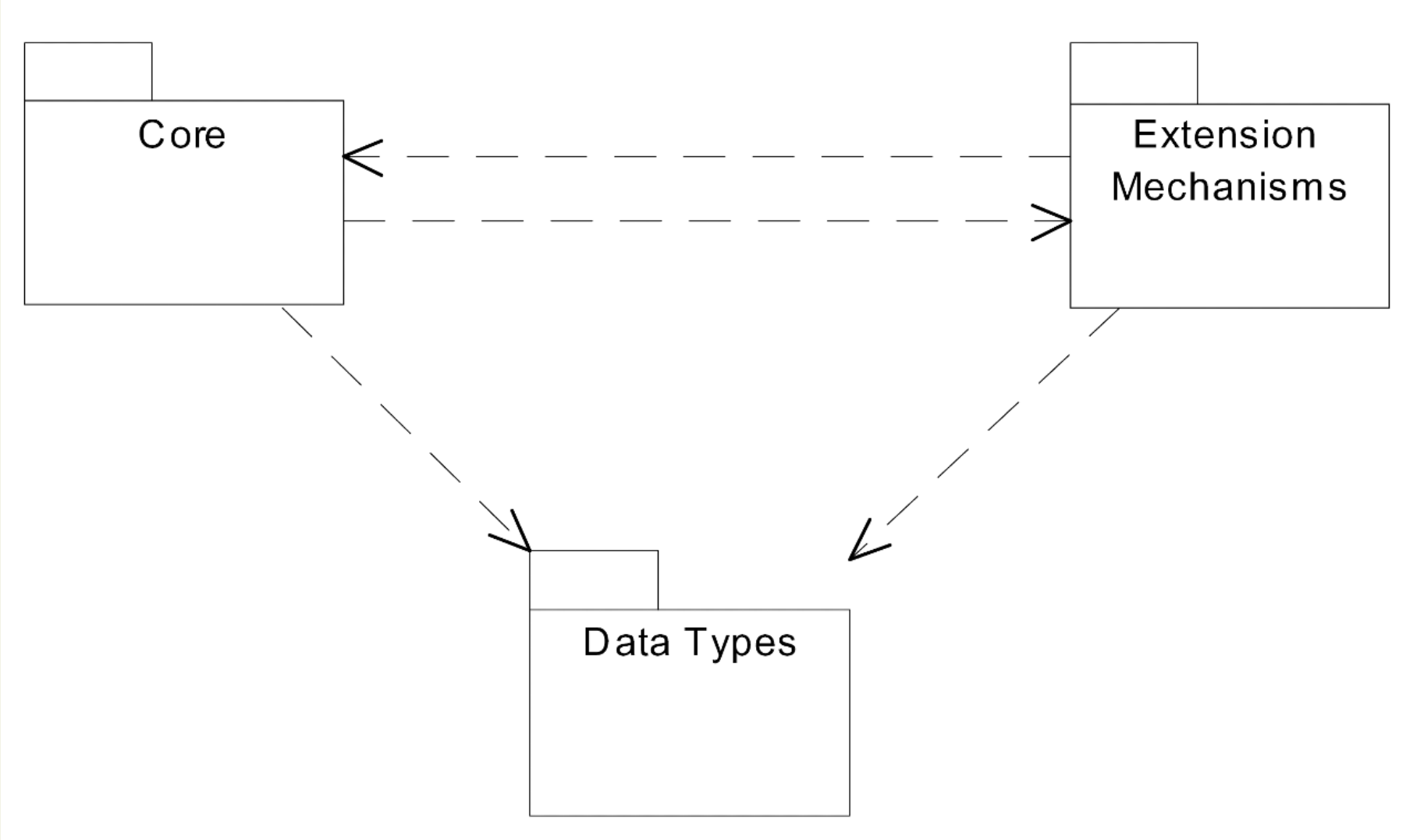


Figure 2-5 Core Package - Backbone

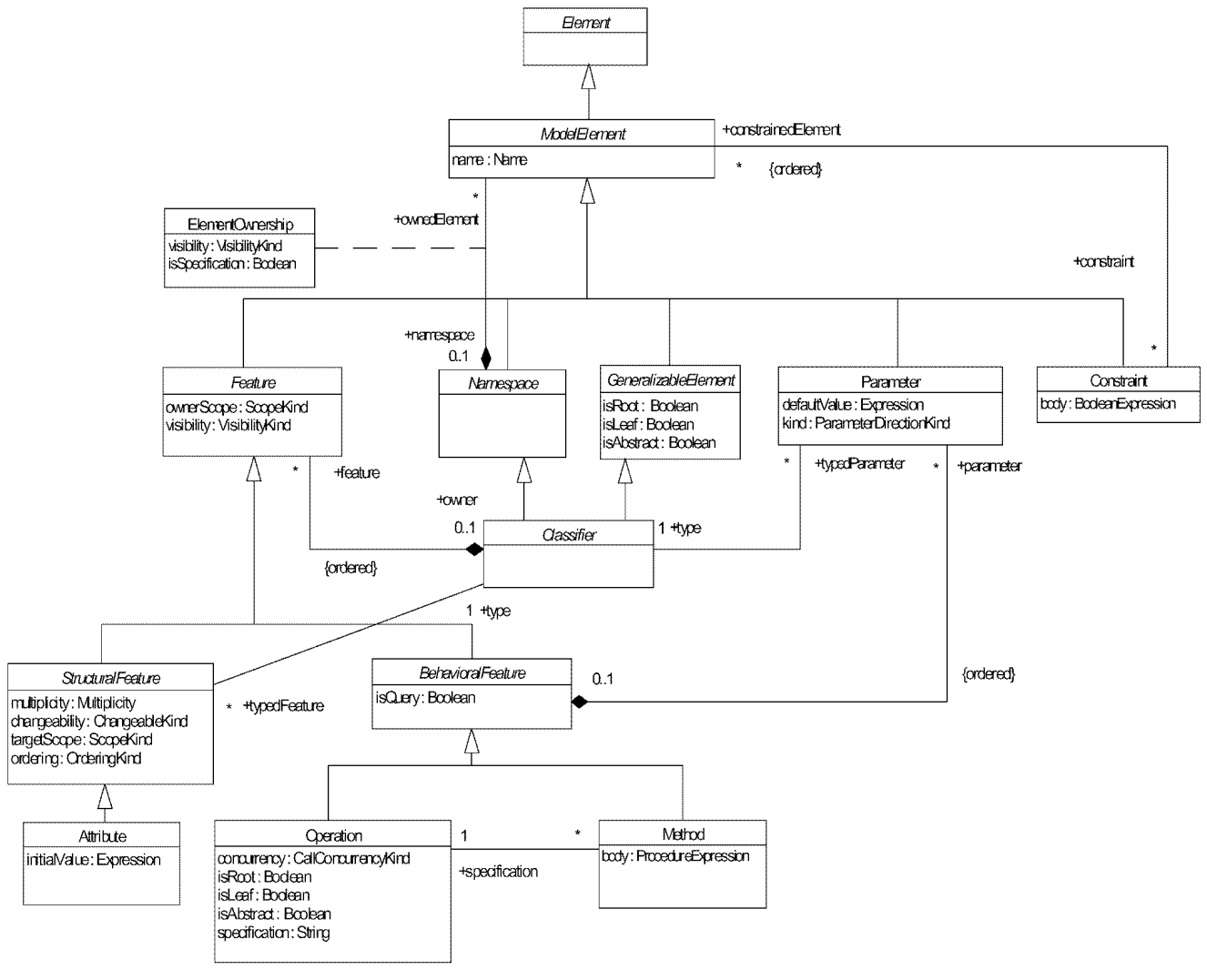


Figure 2-6 Core Package - Relationships

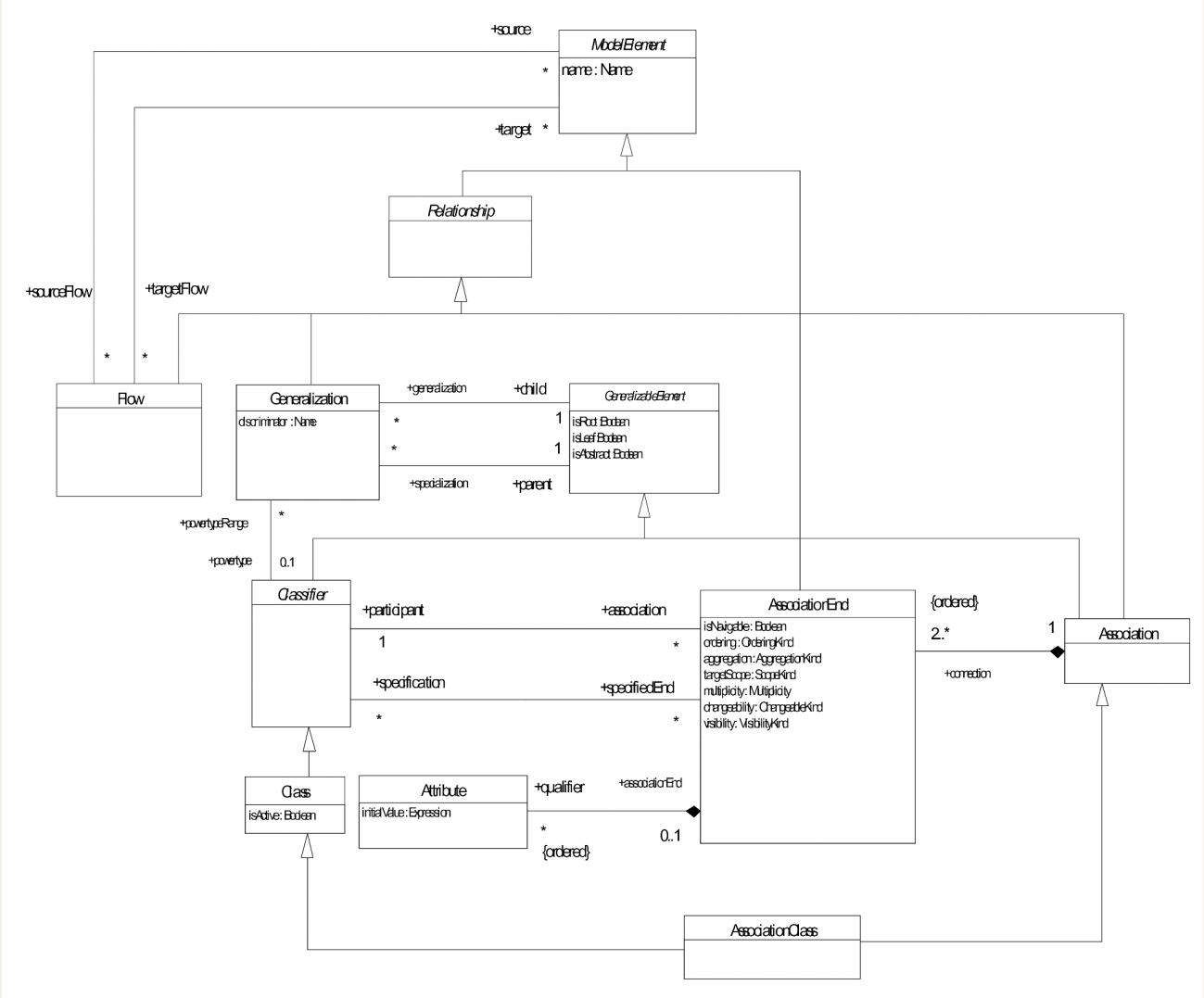


Figure 2-7 Core Package - Dependencies

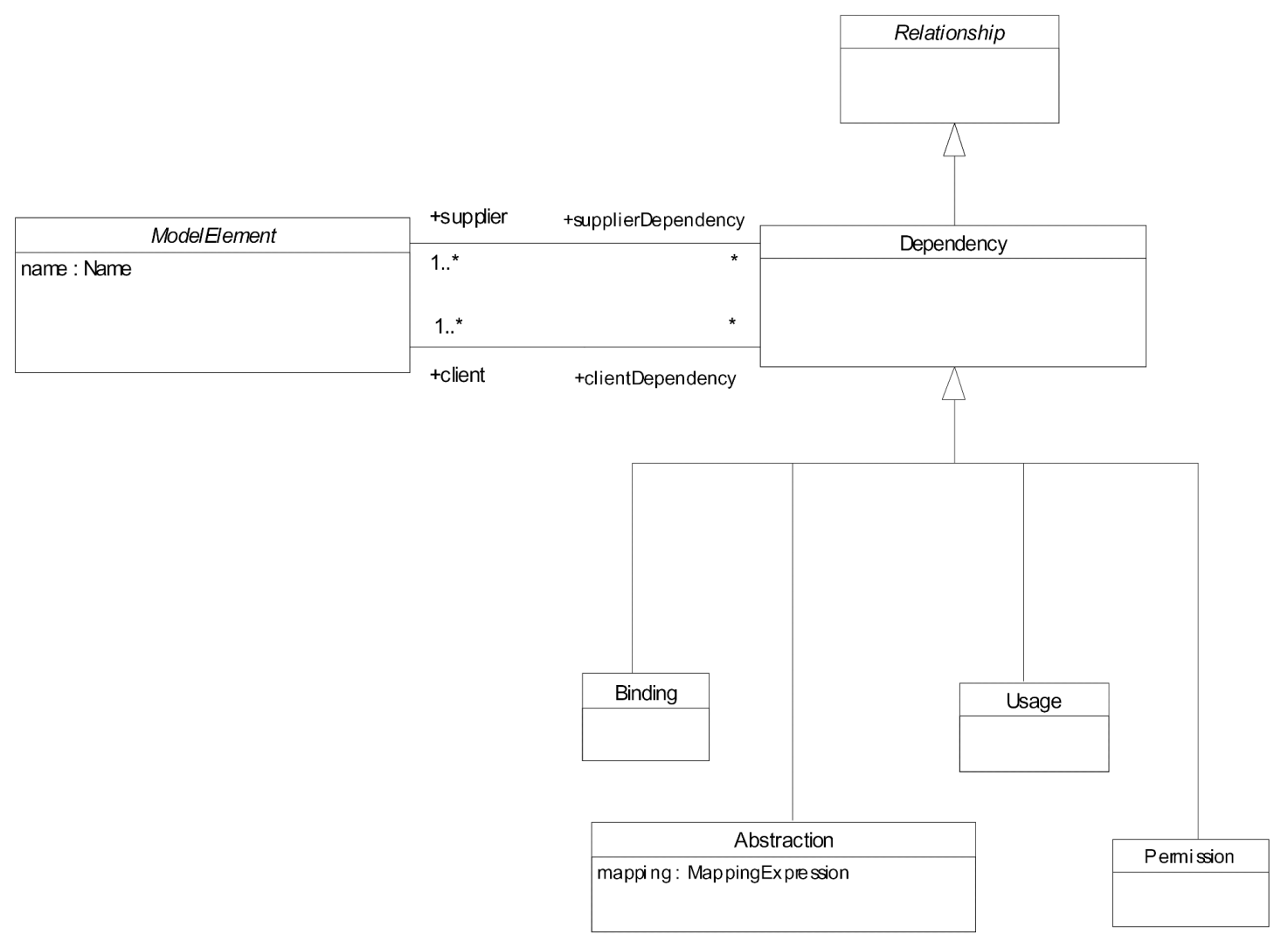


Figure 2-8 Core Package - Classifiers

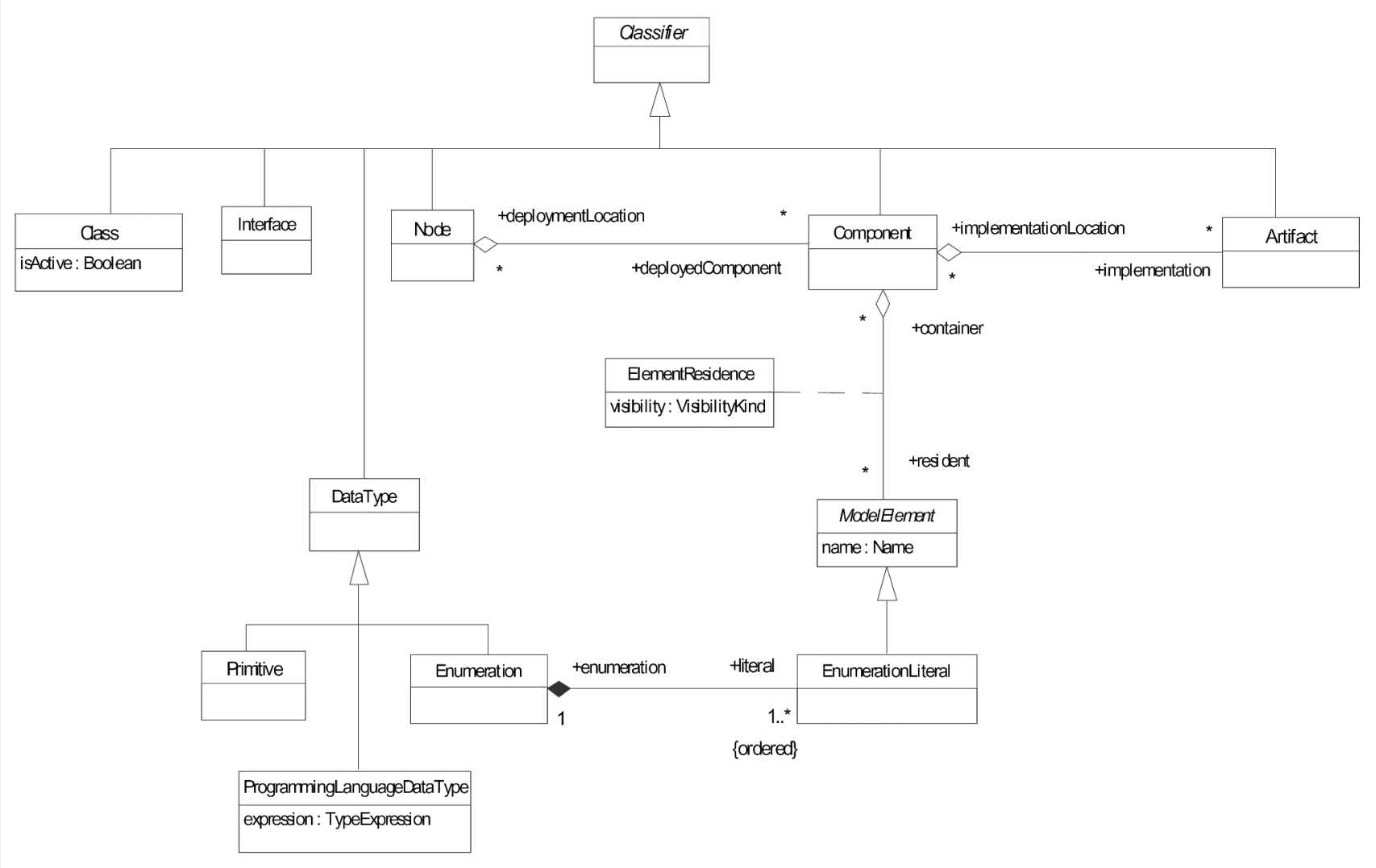
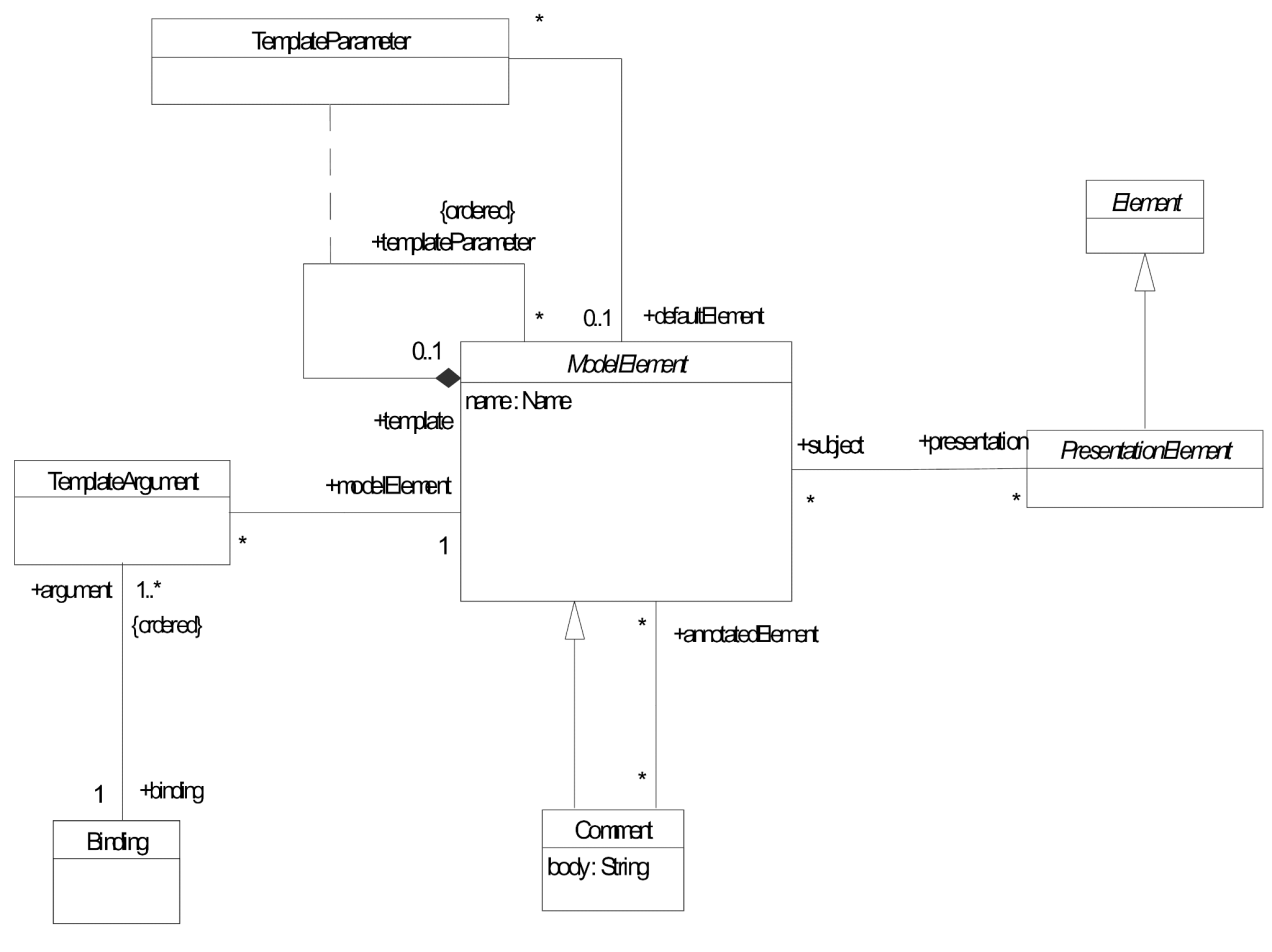


Figure 2-9 Core Package - Auxiliary Elements



8.3 Description for UML Core

Association

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

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

Attributes

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

Associations

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

Stereotypes

implicit	The «implicit» stereotype is applied to an association, specifying that the association is not manifest, but rather is only conceptual.

Standard Constraints

xor	The {xor} constraint is applied to a set of associations, specifying that over that set, exactly one is manifest for each associated instance. Xor is an exclusive or (not inclusive or) constraint.

Tagged Values

persistence	Persistence denotes the permanence of the state of the association, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

Association (cont'd)

Inherited Features

Association is a GeneralizableElement. The following elements are inherited by a child Association.

connection — The child must have the same number of ends as the parent. Each participant class must be a descendant of the participant class in the same position in the parent. If the Association is an AssociationClass, its class properties (attributes, operations, etc.) are inherited. Various other properties are subject to change in the child. This specification is likely to be further clarified in UML 2.0.

Association (cont’d)

Non-Inherited Features

isRoot	Not inheritable by their very nature, but they define the
isLeaf	generalization structure.
isAbstract	
name	Each model element has a unique name.

AssociationClass

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

Inherited Features

AssociationClass inherits features as specified in both Class and Association.

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

AssociationEnd

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

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

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

Attributes

aggregation

When placed on one end (the “target” end), specifies whether the class on the target end is an aggregation with respect to the class on the other end (the “source”end). Only one end can be an aggregation.

Possibilities are:

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

changeability

When placed on one end (the “target” end), specifies whether an instance of the Association may be modified by an instance of the class on the other end (the “source” end). In other words, the attribute controls the access by operations on the class on the opposite end.

Possibilities are:

- changeable - No restrictions on modification.
- frozen - No links may be added by operations on the source class after the creation of the source object. Operations on the target class may add links (provided they are not similarly restricted).
- addOnly - Links may be added at any time by operations on the source object, but once created a link may not be removed by operations on the source class. Operations on the target class may add or remove links (provided they are not similarly restricted).

ordering

When placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered. The ordering must be determined and maintained by Operations that add links. It represents additional information not inherent in the objects or links themselves.

Possibilities are:

- unordered - The links form a set with no inherent ordering.
- ordered - A set of ordered links can be scanned in order.
- Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.

isNavigable	When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. Specification of each direction across the Association is independent. A value of true means that the association can be navigated by the source class and the target rolename can be used in navigation expressions.
multiplicity	When placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given Association.

name	(Inherited from ModelElement) The rolename of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifier; that is, it may be used in the same way as an Attribute and must be unique with respect to Attributes and other pseudo-attributes of the source Classifier.
targetScope	Specifies whether the target value is an instance or a classifier. Possibilities are: • instance. An instance value is part of each link. This is the default. • classifier. A classifier itself is part of each link. Normally this would be fixed at modeling time and need not be stored separately at run time.

visibility

Specifies the visibility of the association end from the viewpoint of the classifier on the other end.

Possibilities are:

- public - Other classifiers may navigate the association and use the rolename in expressions, similar to the use of a public attribute.
- protected - Descendants of the source classifier may navigate the association and use the rolename in expressions, similar to the use of a protected attribute.
- private - Only the source classifier may navigate the association and use the rolename in expressions, similar to the use of a private attribute.
- package - Classifiers in the same package (or a nested subpackage, to any level) as the association declaration may navigate the association and use the rolename in expressions.

Associations

qualifier	An optional list of qualifier Attributes for the end. If the list is empty, then the Association is not qualified.
specification	Designates zero or more Classifiers that specify the Operations that may be applied to an Instance accessed by the AssociationEnd across the Association. These determine the minimum interface that must be realized by the actual Classifier attached to the end to support the intent of the Association. May be an Interface or another Classifier. These classifiers do not indicate the classes of the participants in a link, merely the operations that may be applied when traversing a link.
participant	Designates the Classifier participating in the Association at the given end. A link of the Association contains a reference to an instance of the class (including a descendant of the given class or a class that realizes a given interface) in the given position in the link.
(unnamed composite end)	Designates the Association that owns the AssociationEnd.

Stereotypes

«association»	Specifies a real association (default and redundant, but may be included for emphasis).
«global»	Specifies that the target is a global value that is known to all elements rather than an actual association.
«local»	Specifies that the relationship represents a local variable within a procedure rather than an actual association.
«parameter»	Specifies that the relationship represents a procedure parameter rather than an actual association.
«self»	Specifies that the relationship represents a reference to the object that owns an operation or action rather than an actual association.

Attribute

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

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

Attributes

initialValue	An Expression specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supersede an initial value.)

Associations

associationEnd	Designates the optional AssociationEnd that owns a qualifier attribute. Note that an attribute may be part of an AssociationEnd (in which case it is a qualifier) or part of a Classifier (by inheritance from Feature, in which case it is a feature) but not both. If the value is empty, the attribute is not a qualifier attribute.

BehavioralFeature

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

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

Attributes

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

Associations

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

Stereotypes

«create»	Specifies that the designated feature creates an instance of the classifier to which the feature is attached. May be promoted to the Classifier containing the feature.
«destroy»	Specifies that the designated feature destroys an instance of the classifier to which the feature is attached. May be promoted to the classifier containing the feature.

Class

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

In the metamodel, a Class describes a set of Objects sharing a collection of Features, including Operations, Attributes and Methods, that are common to the set of Objects. Furthermore, a Class may realize zero or more Interfaces; this means that its full descriptor (see Section 2.5.4.4, Inheritance, on page 2-70 for the definition) must contain every Operation from every realized Interface (it may contain additional operations as well).

Class (cont'd)

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

Attributes

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

Stereotypes

«auxiliary»	Specifies a class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined explicitly using a Focus class or implicitly by a dependency relationship. Auxiliary classes are typically used together with Focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: «focus».
«focus»	Specifies a class that defines the core logic or control flow for one or more auxiliary classes that support it. Support classes may be defined explicitly using Auxiliary classes or implicitly by dependency relationships. Focus classes are typically used together with one or more Auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: «auxiliary».

«implementation»	Specifies the implementation of a class in some programming language (for example, C++, Smalltalk, Java) in which an instance may not have more than one class. This is in contrast to Class, for which an instance may have multiple classes at one time and may gain or lose classes over time, and an object (a child of instance) may dynamically have multiple classes. An Implementation class is said to *realize* a Type if it provides all of the operations defined for the Type with the same behavior as specified for the Type's operations. An Implementation Class may realize a number of different Types. Note that the physical attributes and associations of the Implementation class do not have to be the same as those of any Type it realizes and that the Implementation Class may provide methods for its operations in terms of its physical attributes and associations. See also: «type».

«type»	Specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods, maintain its own thread of control, or be nested. However, it may have attributes and associations. The associations of a Type are defined solely for the purpose of specifying the behavior of the type's operations and do not represent the implementation of state data. Although an object may have at most one Implementation Class, it may conform to multiple different Types. See also: «implementation».

Class (cont'd)

Inherited Features

Class is a GeneralizableElement. The following elements are inherited by a child classifier, in addition to those specified under its parent, Classifier.

isActive	The child may be active when the parent is passive, but not vice versa. In most cases, they are the same.

Classifier

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

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

Classifier (cont'd)

Classifier is a child of GeneralizableElement and Namespace. As a GeneralizableElement, it may inherit Features and participation in Associations (in addition to things inherited as a ModelElement). It also inherits ownership of StateMachines, Collaborations, etc.

As a Namespace, a Classifier may declare other Classifiers nested in its scope. Nested Classifiers may be accessed by other Classifiers only if the nested Classifiers have adequate visibility. There are no data value or state consequences of nested Classifiers (i.e., it is not an aggregation or composition).

Associations

feature	An ordered list of Features, like Attribute, Operation, Method, owned by the Classifier.
association	Denotes the AssociationEnd of an Association in which the Classifier participates at the given end. This is the inverse of the participant association from AssociationEnd. A link of the association contains a reference to an instance of the class in the given position.
powertypeRange	Designates zero or more Generalizations for which the Classifier is a powertype. If the cardinality is zero, then the Classifier is not a powertype; if the cardinality is greater than zero, then the Classifier is a powertype over the set of Generalizations designated by the association, and the child elements of the Generalizations are the instances of the Classifier as a powertype. A Classifier that is a powertype can be marked with the «powertype» stereotype.
specifiedEnd	Indicates an AssociationEnd for which the given Classifier specifies operations that may be applied to instances obtained by traversing the association from the other end. (This relationship does not define the structure of the association, merely operations that may be applied on traversing it.)

Stereotypes

«metaclass»	Specifies that the instances of the classifier are classes.
«powertype»	Specifies that the classifier is a metaclass whose instances are siblings marked by the same discriminator. For example, the metaclass TreeSpecies might be a power type for the subclasses of Tree that represent different species, such as AppleTree, BananaTree, and CherryTree.
«process»	Specifies a classifier that represents a heavy-weight flow of control.
«thread»	Specifies a classifier that represents a flow of control.
«utility»	Specifies a classifier that has no instances, but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped.

Tagged Values

persistence	Persistence denotes the permanence of the state of the classifier, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
semantics	Semantics is the specification of the meaning of the classifier.

Classifier (cont'd)

Inherited Features

Classifier is a GeneralizableElement. The following elements are inherited by a child classifier. Note that inheritance makes the inherited elements part of the (virtual) full descriptor of the classifier, but it does not change its actual data structure. See the explanation for the meaning of each kind of inheritance.

associationEnd	The child class inherits participation in all associations of its parent, subject to all the same properties.
constraint	Constraints on the parent apply to the child.

Classifier (cont'd)

feature — Attributes of the parent are part of the full descriptor of the child and may not be declared again or overridden. Operations of the parent are part of the full descriptor of the child but may be overridden; a redeclaration may change its hierarchy location (isRoot, isLeaf, isAbstract) but may not change its specification or parameter structure. The concurrency level may be loosened (e.g., from guarded to concurrent). An overriding operation may link to a different Method. An overriding operation can have isQuery=true when the parent had a false value, but not vice versa (in other words, once a side-effect, always a sideeffect).

Classifier (cont'd)

feature(cont'd)	Methods of the parent are part of the full descriptor of the child but may be overridden. An overriding method can set the isQuery status, change its hierarchy structure, but may not change its parameter structure. It may link to a different operation that overrides the operation of the parent method.

Classifier (cont'd)

generalization specialization	These are implicitly inherited, in the sense that they define ancestors and descendants, but not explicitly inherited, as they are the arcs in the generalization graph. They establish the generalization structure itself as a directed graph, into which the child classifier fits.
ownedElement	The namespace of the parent is available to the child, except for private access.

Classifier (cont'd)

Non-Inherited Features

The following elements are not inherited by a child classifier:

isRoot	By their very nature, these are not inherited.
isLeaf	
isAbstract	
name	Each classifier has its own unique name.
parameter	Template structure is not inherited. Each classifier must declare its own template structure, if any. A nontemplate can be child of a template and vice versa.

Classifier (cont’d)

powertypeRange	A powertype corresponds to a particular node in the generalization hierarchy, so it is not inherited.

Constraint

A constraint is a semantic condition or restriction expressed in text.

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

Attributes

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

Associations

constrainedElement	A ModelElement or list of ModelElements affected by the Constraint. If the constrained element is a Stereotype, then the constraint applies to all ModelElements that use the stereotype.

Stereotypes

«invariant»	Specifies a constraint that must be attached to a set of classifiers or relationships. It indicates that the conditions of the constraint must hold over time (for the time period of concern in the particular containing element) for the classifiers or relationships and their instances.
«postcondition»	Specifies a constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold after the invocation of the operation.
«precondition»	Specifies a constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold for the invocation of the operation.
«stateInvariant»	Specifies a constraint that must be attached to a state vertex in a state machine that has a classifier for a context. The stereotype indicates that the constraint holds for instances of the classifier when an instance is in that state.

Feature

A feature is a property, like operation or attribute, which is encapsulated within a Classifier.

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

Attributes

name	(Inherited from ModelElement) The name used to identify the Feature within the Classifier or Instance. It must be unique across inheritance of names from ancestors including names of outgoing AssociationEnd. See more specific rules for the exact details. Attributes, discriminators, and opposite association ends must have unique names in the set of inherited names. There may be multiple declarations of the same operation. Multiple operations may have the same name but different signatures; see the rules for precise details.
ownerScope	Specifies whether Feature appears in each Instance of the Classifier or whether there is just a single instance of the Feature for the entire Classifier. Possibilities are: • instance - Each Instance of the Classifier holds its own value for the Feature. • classifier - There is just one value of the Feature for the entire Classifier.

visibility	Specifies whether the Feature can be used by other Classifiers. Visibilities of nested Classifiers combine so that the most restrictive visibility is the result. Possibilities include: • public - Any outside Classifier with visibility to the Classifier can use the Feature. • protected - Any descendent of the Classifier can use the Feature. • private - Only the Classifier itself can use the Feature. • package - Any Classifier declared in the same package (or a nested subpackage, to any level) as the owner of the Feature can use the Feature.

Associations

owner	The Classifier declaring the Feature. Note that an Attribute may be owned by a Classifier (in which case it is a feature) or an AssociationEnd (in which case it is a qualifier) but not both.

Operation

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

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

Attributes

concurrency	Specifies the semantics of concurrent calls to the same passive instance; that is, an Instance originating from a Classifier with isActive=false. Active instances control access to their own Operations so this property is usually (although not required in UML) set to sequential. Possibilities include: • sequential - Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.

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

isAbstract	If true, then the operation does not have an implementation, and one must be supplied by a descendant. If false, the operation must have an implementation in the class or inherited from an ancestor.
isLeaf	If true, then the implementation of the operation may not be overriden by a descendant class. If false, then the implementation of the operation may be overridden by a descendant class (but it need not be overridden).
isRoot	If true, then the class must not inherit a declaration of the same operation. If false, then the class may (but need not) inherit a declaration of the same operation. (But the declaration must match in any case; a class may not modify an inherited operation declaration.)

Tagged Values

semantics	Semantics is the specification of the meaning of the operation.

StructuralFeature

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

In the metamodel, a StructuralFeature declares a structural aspect of an Instance of a Classifier, such as an Attribute. For example, it specifies the multiplicity and changeability of the StructuralFeature. StructuralFeature is an abstract metaclass.

Attributes

changeability	Whether the value may be modified after the object is created. Possibilities are: • changeable - No restrictions on modification. • frozen - The value may not be altered after the object is instantiated and its values initialized. No additional values may be added to a set. • addOnly - Meaningful only if the multiplicity is not fixed to a single value. Additional values may be added to the set of values, but once created a value may not be removed or altered.
multiplicity	The possible number of data values for the feature that may be held by an instance. The cardinality of the set of values is an implicit part of the feature. In the common case in which the multiplicity is 1..1, then the feature is a scalar; that is, it holds exactly one value.

ordering

Specifies whether the set of instances is ordered. The ordering must be determined and maintained by Operations that add values to the feature. This property is only relevant if the multiplicity is greater than one.

Possibilities are:

- unordered - The instances form a set with no inherent ordering.
- ordered - A set of ordered instances can be scanned in order.
- Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.

targetScope

Specifies whether the targets are ordinary Instances or are Classifiers.

Possibilities are:

- instance - Each value contains a reference to an Instance of the target Classifier. This is the setting for a normal Attribute.
- classifier - Each value contains a reference to the target Classifier itself. This represents a way to store meta-information.

Associations

type	Designates the classifier whose instances are values of the feature. Must be a Class, Interface, or DataType. The actual type may be a descendant of the declared type or (for an Interface) a Class that realizes the declared type.

Tagged Values

persistence	Persistence denotes the permanence of the state of the feature, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

8.4 Well-formedness Rules for UML Core

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

Association

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

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

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

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

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

```
self.allConnections->size >=3 implies
       self.allConnections->forall(aggregation = #none)
```

Association (cont'd)

[4] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association, or be Classifiers with public visibility in other Namespaces to which the Namespace of the Association has access Permissions.

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

    self.allConnections->forAll(r | self.namespace.allContents->excludes
    (r.participant) implies

       self.namespace.clientDependency->exists (d |
          d.oclIsTypeOf(Permission) and
          d.stereotype.name = 'access' and
          d.supplier.oclAsType(Namespace).ownedElement->select (e |
                e.elementOwnership.visibility =
                   #public)->includes (r.participant) or
          d.supplier.oclAsType(GeneralizableElement).
             allParents.oclAsType(Namespace).ownedElement->select (e |
                e. elementOwnership.visibility =
                   #public)->includes (r.participant) or
          d.supplier.oclAsType(Package).allImportedElements->select (e |
                e. elementImport.visibility =
                   #public) ->includes (r.participant) ) )
```

Association (cont’d)

Additional operations

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

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

AssociationClass

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

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

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

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

Additional operations

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

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

AssociationEnd

[1] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end.

```
(self.participant.oclIsKindOf (Interface) or
self.participant.oclIsKingOf (DataType)) implies
        self.association.connection->select
             (ae | ae <> self)->forAll(ae | ae.isNavigable = #false)
```

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

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

Attribute

No extra well-formedness rules.

BehavioralFeature

[1] All Parameters should have a unique name.

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

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

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

Additional operations

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

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

[2] The operation matchesSignature checks if the argument has a signature that would clash with the signature of the instance itself (and therefore must be unique). Mismatches in kind or any differences in return parameters do not

cause a mismatch:

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

Class

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

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

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

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

Classifier

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

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

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

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

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

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

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

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( a |
```

```
        not self.allOppositeAssociationEnds->union (self.allContents)->collect ( q |
            q.name )->includes (a.name) )
```

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

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

[6] For each Operation in an specification realized by the Classifier, the Classifier must have a matching Operation.

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

[7] All of the generalizations in the range of a powertype have the same discriminator.

```
self.powertypeRange->forAll
        (g1, g2 | g1.discriminator = g2.discriminator)
```

[8] Discriminator names must be distinct from attribute names and opposite AssociationEnd names.

```
self.allDiscriminators->intersection (self.allAttributes.name->union
(self.allOppositeAssociationEnds.name))->isEmpty
```

Additional operations

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

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

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

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

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

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

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

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

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

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

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

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

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

```
oppositeAssociationEnds : Set (AssociationEnd);
```

```
oppositeAssociationEnds =
       self.associations->select ( a | a.connection->select ( ae |
           ae.participant = self ).size = 1 )->collect ( a |
               a.connection->
                   select ( ae | ae.participant <> self ) )->union (
       self.associations->select ( a | a.connection->select ( ae |
           ae.participant = self ).size > 1 )->collect ( a |
               a.connection) )
```

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

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

[9] The operation specification yields the set of Classifiers that the current Classifier realizes.

```
specification: Set(Classifier)
specification = self.clientDependency->
       select(d |
           d.oclIsKindOf(Abstraction)
           and d.stereotype.name = "realization"
           and d.supplier.oclIsKindOf(Classifier))
       .supplier.oclAsType(Classifier)
```

[10] The operation allContents returns a Set containing all ModelElements contained in the Classifier together with the contents inherited from its parents.

```
allContents : Set(ModelElement);
```

```
allContents = self.contents->union(
       self.parent.allContents->select(e |
           e.elementOwnership.visibility = #public or
           e.elementOwnership.visibility = #protected))
```

[11] The operation allDiscriminators results in a Set containing all Discriminators of the Generalizations from which the Classifier is descended itself and all its inherited Features.

```
allDiscriminators : Set(Name);
allDiscriminators = self.generalization.discriminator->union(
self.parent.oclAsType(Classifier).allDiscriminators)
```

Constraint

[1] A Constraint cannot be applied to itself.

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

Feature

No extra well-formedness rules.

Operation

No additional well-formedness rules.

StructuralFeature

[1] The connected type should be included in the owner s Namespace.

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

[2] The type of a StructuralFeature must be a Class, DataType, or Interface.

```
self.type.oclIsKindOf(Class) or
self.type.oclIsKindOf(DataType) or
self.type.oclIsKindOf(Interface)
```

8.5 ”Detailed Semantics” for UML Core

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

Association

An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association ends, each specifying a connected classifier and a set of properties that must be fulfilled for the relationship to be valid. The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A multiplicity is a range of nonnegative integers. The association end also states whether or not the connection may be traversed towards the instance playing that role in the connection (isNavigable), for instance, if the instance is directly reachable via the association. An association-end also specifies whether or not an instance playing that role in a connection may be

replaced by another instance.

It may state

- that no constraints exist (changeable),
- that the link cannot be modified once it has been initialized (frozen), or
- that new links of the association may be added but not removed or altered (addOnly).

These constraints do not affect the modifiability of the objects themselves that are attached to the links. Moreover, the classifier, or (a child of) the classifier itself. The ordering attribute of association-end states that if the instances related to a single instance at the other end have an ordering that must be preserved, the order of insertion of new links must be specified by operations

that add or modify links. Note that sorting is a performance optimization and is not an example of a logically ordered association, because the ordering information in a sort does not add any information.

In UML, Associations can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) shareable aggregate. Since the aggregate construct can have several different meanings depending on the application area, UML gives a more precise meaning to two of these constructs; that is, association and composite aggregate and leaves the shareable aggregate more loosely defined in between.

An association may represent an aggregation; that is, a whole/part relationship. In this case, the association-end attached to the whole

element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of the parts. In implementation terms, it is responsible for their memory allocation. If a composite object is destroyed, it must destroy all of its parts. It may remove a part and give it to another composite object, which then assumes responsibility for it. If the multiplicity from a part to composite is zero-to-one, the composite may remove the part, and the part may assume responsibility for itself, otherwise it may not live apart from a composite.

A consequence of these rules is that a composite implies propagation semantics; that is, some of the dynamic semantics of the whole is propagated to its parts. For example, if the whole is copied or destroyed, then so are the parts as well (because a part may belong to at most one composite).

A classifier on the composite end of an association may have parts that are classifiers and associations. At the instance level, an instance of a part element is considered part of the instance of a composite element. If an association is part of a composite and it connects two classes that are also part of the same composite, then a link of the association will connect objects that are part of the same composite object of which the link is part.

A shareable aggregation denotes weak ownership; that is, the part

may be included in several aggregates and its owner may also change over time. However, the semantics of a shareable aggregation does not imply deletion of the parts when an aggregate referencing it is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship; that is, the instances form a directed, non-cyclic graph. Composition instances form a strict tree (or rather a forest).

A qualifier declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A qualifier instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is

unique with respect to the qualified object, and designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences. In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value.

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

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

AssociationClass

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

The AssociationClass construct can be expressed in a few different ways in the metamodel (for example, as a subclass of Class, as a subclass of Association, or as a subclass of Classifier). Since an AssociationClass is a construct being both an association (having a

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

The terms child, subtype, and subclass are synonyms and mean that an instance of a classifier being a subtype of another classifier can always be used where an instance of the latter classifier is expected. The neutral terms parent and child, with the transitive closures ancestor and descendant, are the preferred terms in this document.

Class

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

When a class is instantiated to create a new object, a new instance is created, which is initialized containing an attribute value for each

attribute found in the full class descriptor. The object is also initialized with a connection to the list of methods in the full class descriptor.

Note – An actual implementation behaves as if there were a full class descriptor, but many clever optimizations are possible in practice.

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

A class can have generalizations to other classes. This means that the full class descriptor of a class is derived by inheritance from its own segment declaration and those of its ancestors. Generalization between classes implies substitutability; that is, an instance of a class may be used whenever an instance of a superclass is expected. If the class is specified as a root, it cannot be a subclass of

other classes. Similarly, if it is specified as a leaf, no other class can be a subclass of the class.

Each attribute declared in a class has a visibility and a type. The visibility defines if the attribute is publicly available to any class, if it is only available inside the class and its subclasses (protected), if it can be used within the containing package (package), or if it can only be used inside the class (private). The targetScope of the attribute declares whether its value should be an instance (of a child) of that type or if it should be (a child of) the type itself.

There are two alternatives for the ownerScope of an attribute:

- it may state that each object created by the class (or by its subclasses) has its own value of the attribute, or

- that the value is owned by the class itself.

An attribute also declares how many attribute values should be connected to each owner (multiplicity), what the initial values should be, and if these attribute values may be changed to:

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

For each operation, the operation name, the types of the parameters, and the return type(s) are specified, as well as its visibility (see above). An operation may also include a specification of the effects of its invocation. The specification can be done in several different ways (for example, with pre- and post-conditions, pseudo-code, or

just plain text). Each operation declares if it is applicable to the instances, the class, or to the class itself (ownerScope). Furthermore, the operation states whether or not its application will modify the state of the object (isQuery). The operation also states whether or not the operation may be realized by a different method in a subclass (isPolymorphic). A method realizing an operation has the same signature as the operation and a body implementing the specification of the operation. Methods in descendants override and replace methods inherited from ancestors (see Section 2.5.4.4, Inheritance, on page 2-70). Each method implements an operation declared in the class or inherited from an ancestor. The same operation may be declared more than once in a full class descriptor, but their descriptions must all match, except that the generalization properties (isRoot, IsAbstract, isLeaf) may vary, and a child operation

may strengthen query properties (the child may be a query even though the parent is not). The specification of the method must match the specification of its matching operation, as defined above for operations. Furthermore, if the isQuery attribute of an operation is true, then it must also be true in any realizing method. However, if it is false in the operation, it may still be true in the method if the method does not actually modify the state to carry out the behavior required by the operation (this can only be true if the operation does not inherently modify state). The visibility of a method must match its operation.

Classes may have associations to each other. This implies that objects created by the associated classes are semantically connected; that is, that links exist between the objects, according to the requirements of the associations. See Association on the next

page. Associations are inherited by subclasses.

A class may realize a set of interfaces. This means that each operation found in the full descriptor for any realized interface must be present in the full class descriptor with the same specification (see Section 2.5.4.4, Inheritance, on page 2-70). The relationship between interface and class is not necessarily one-to-one; a class may offer several interfaces and one interface may be offered by more than one class. The same operation may be defined in multiple interfaces that a class supports; if their specifications are identical then there is no conflict; otherwise, the model is ill formed. Moreover, a class may contain additional operations besides those found in its interfaces.

A class acts as the namespace for various kinds of contained

elements defined within its scope including classes, interfaces, and associations (note that this is purely a scoping construction and does not imply anything about aggregation), the contained classifiers can be used as ordinary classifiers in the container class. If a class inherits another class, the contents of the ancestor are available to its descendants if the visibility of an element is public or protected; however, if the visibility is private, then the element is not visible and therefore not available in the descendant.

Inheritance

To understand inheritance it is first necessary to understand the concept of a full descriptor and a segment descriptor. A full descriptor is the full description needed to describe an object or other instance (see Section 2.5.4.5, Instantiation, on page 2-71). It contains a description of all of the attributes, associations, and operations that the object contains. In a pre-object-oriented language, the full descriptor of a data structure was declared directly in its entirety. In an object-oriented language, the description of an object is built out of incremental segments that are combined using inheritance to produce a full descriptor for an object. The segments are the modeling elements that are actually declared in a model. They include elements such as class and other generalizable elements. Each generalizable element contains a list of features and

other relationships that it adds to what it inherits from its ancestors. The mechanism of inheritance defines how full descriptors are produced from a set of segments connected by generalization. The full descriptors are implicit, but they define the structure of actual instances.

Each kind of generalizable element has a set of inheritable features. For any model element, these include constraints. For classifiers, these include features (attributes, operations, signal receptions, and methods) and participation in associations. The ancestors of a generalizable element are its parents (if any) together with all of their ancestors (with duplicates removed). For a Namespace (such as a Package or a Class with nested declarations), the public or protected contents of the Namespace are available to descendants of the Namespace.

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

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

The purpose of the full descriptor is explained under Section 2.5.4.5, Instantiation, on page 2-71.

Instantiation

The purpose of a model is to describe the possible states of a system and their behavior. The state of a system comprises objects, values, and links. Each object is described by a full class descriptor. The class corresponding to this descriptor is the direct class of the object. If an object is not completely described by a single class (multiple classification), then any class in the minimal set of unrelated (by generalization) classes whose union completely describes the object is a direct class of the object. Similarly each link has a direct association and each value has a direct data type. Each of these instances is said to be a direct instance of the classifier from which its full descriptor was derived. An instance is an indirect instance of the classifier or any of its ancestors.

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

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

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

Interface

The purpose of an interface is to collect a set of operations that constitute a coherent service offered by classifiers. Interfaces provide a way to partition and characterize groups of operations. An interface is only a collection of operations with a name. It cannot be directly instantiated. Instantiable classifiers, such as class or use case, may use interfaces for specifying different services offered by their instances. Several classifiers may realize the same interface. All of them must contain at least the operations matching those contained in the interface. The specification of an operation contains the signature of the operation (i.e., its name, the types of the parameters, and the return type). An interface does not imply any internal structure of the realizing classifier. For example, it does not define which algorithm to use for realizing an operation. An operation may,

however, include a specification of the effects of its invocation. The specification can be done in several different ways (e.g., with pre and post-conditions, pseudo-code, or just plain text).

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

An interface can be a child of other interfaces denoted by generalizations. This means that a classifier offering the interface must provide not only the operations declared in the interface but also those declared in the ancestors of the interface. If the interface

is specified as a root, it cannot be a child of other interfaces. Similarly, if it is specified as a leaf, no other interface can be a child of the interface.

Operation

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

PresentationElement

The responsibility of presentation element is to provide a textual and graphical projection of a collection of model elements. In this context, projection means that the presentation element represents a human readable notation for the corresponding model elements. The notation for UML can be found in Chapter 3 of this document.

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

Template

A template is a parameterized model element that cannot be used directly in a model. Instead, it may be used to generate other model elements using the Binding relationship; those generated model elements can be used in normal relationships with other elements.

A template represents the parameterization of a model element, such as a class or an operation, although conceptually any model element may be used (but not all may be useful). The template element is attached by composite aggregation to an ordered list of parameter elements. Each parameter element has a name that represents a parameter name within the template element. Any use of the name within the scope of the template element represents an unbound parameter that is to be replaced by an actual value in a Binding of the

template. For example, a parameter may represent the type of an attribute of a class (for a class template). The corresponding attribute would have an association to the template parameter as its type.

Note that the scope of the template includes all of the elements recursively owned by it through composite aggregation. For example, a parameterized class template owns its attributes, operations, and so on. Neither the parameterized elements nor its contents may be used directly in a model without binding.

A template element has the templateParameter association to a list of ModelElements that serve as its parameters. To avoid introducing metamodel (M2) elements in an ordinary (M1) model, the model contains a representative of each parameter element, rather than the type of the parameter element. For example, a frequent kind of parameter is a class. Instead of including the metaclass Class in the

(M1) ordinary model, a dummy class must be declared whose name is the name of the parameter. This dummy element is meaningful only within the template (it may not be used within the wider model) and it has no features (such as attributes and operations), because the features are part of an actual element that is supplied when the template is bound. Because a template parameter is only a dummy that lacks internal structure, it may violate well-formedness constraints of elements of its kind; the actual elements supplied during binding must satisfy ordinary well-formedness constraints.

Note also that when the template is bound, the bound element does not show the explicit structure of an element of its kind; it is a stub. Its semantics and wellformedness rules must be evaluated as if the actual substitutions of actual elements for parameters had been made; but the expansions are not explicitly shown in a canonical

model as they are regarded as derived.

A template element is therefore effectively isolated from the directly-usable part of the model and is indirectly connected to its ultimate instances through Binding associations to bound elements. The bound elements may be used in ordinary models in places where the model element underlying the template could be used.

Miscellaneous

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

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

A Usage or Binding dependency can be established only between elements in the same model, since the semantics of a model cannot

be dependent on the semantics of another model. If a connection is to be established between elements in different models, a Trace or Refinement should be used. Refinement can connect elements in different or same models.

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

A data type is a special kind of classifier, similar to a class, but whose instances are primitive values (not objects). For example, the integers and strings are usually treated as primitive values. A

primitive value does not have an identity, so two occurrences of the same value cannot be differentiated. Usually, it is used for specification of the type of an attribute. An enumeration type is a user-definable type comprising a finite number of values.

8.6 Other Class Diagrams for UML Metamodel

Figure 2-10 Extension Mechanisms

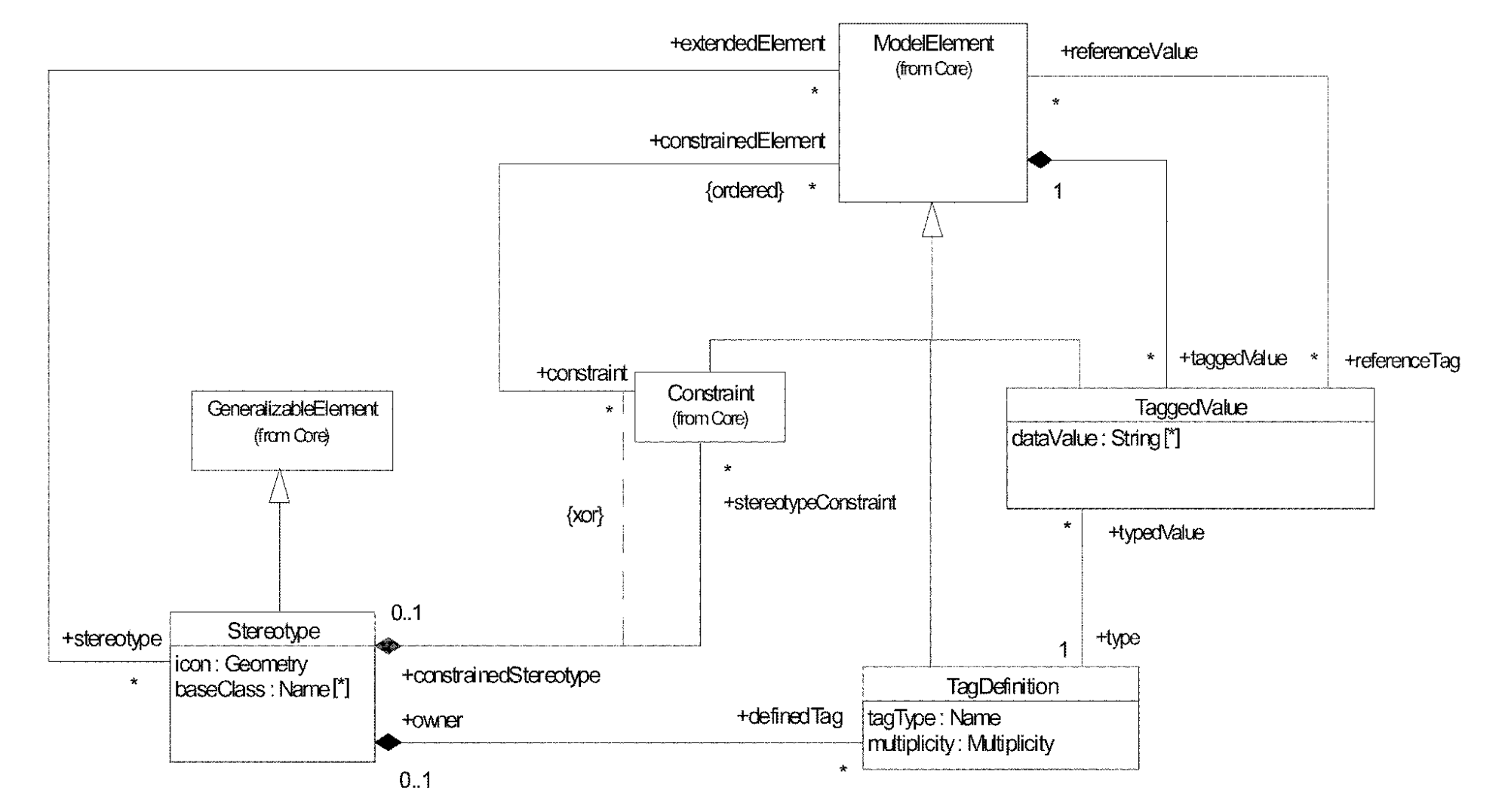


Figure 2-11 Data Types Package - Main

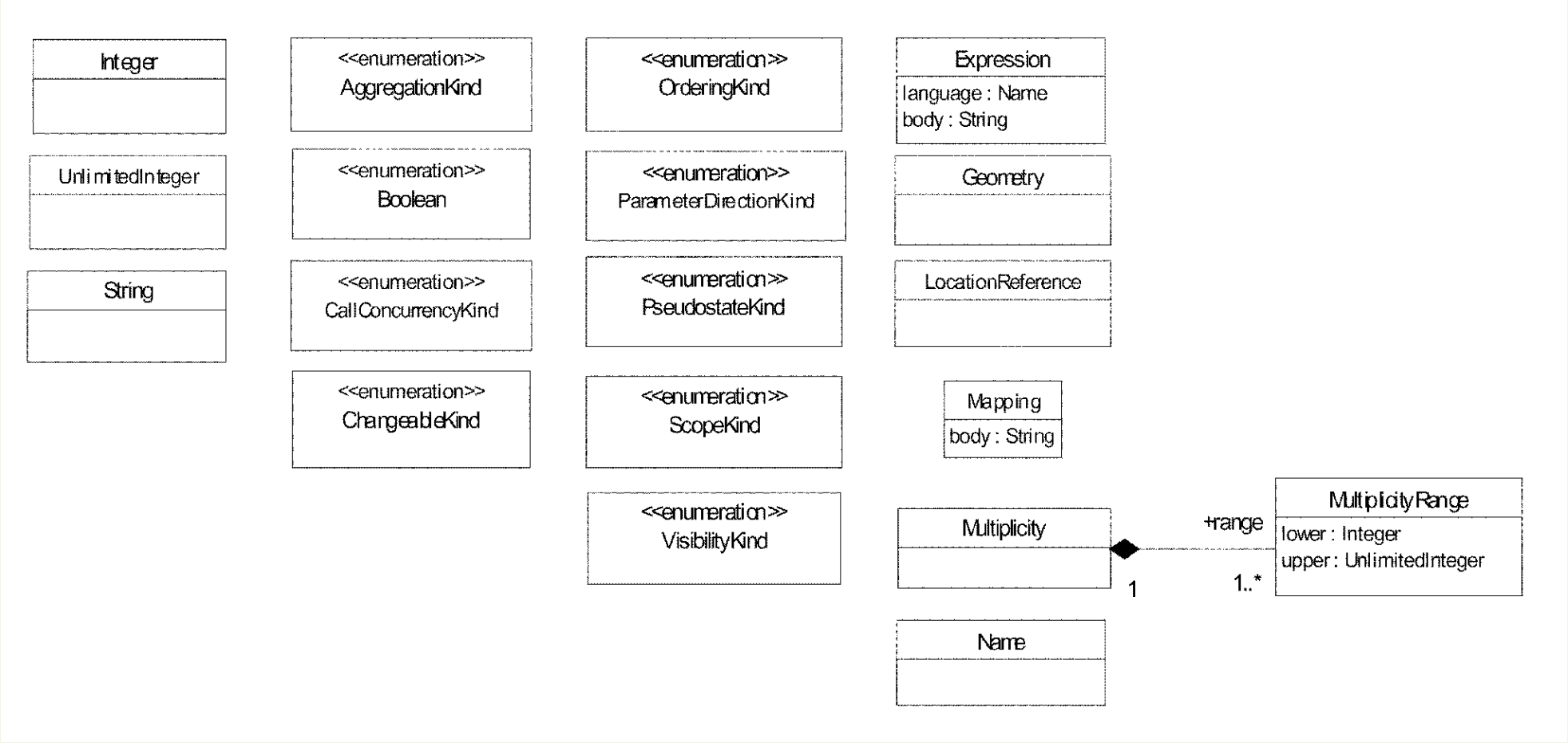


Figure 2-12 Data Types Package - Expressions

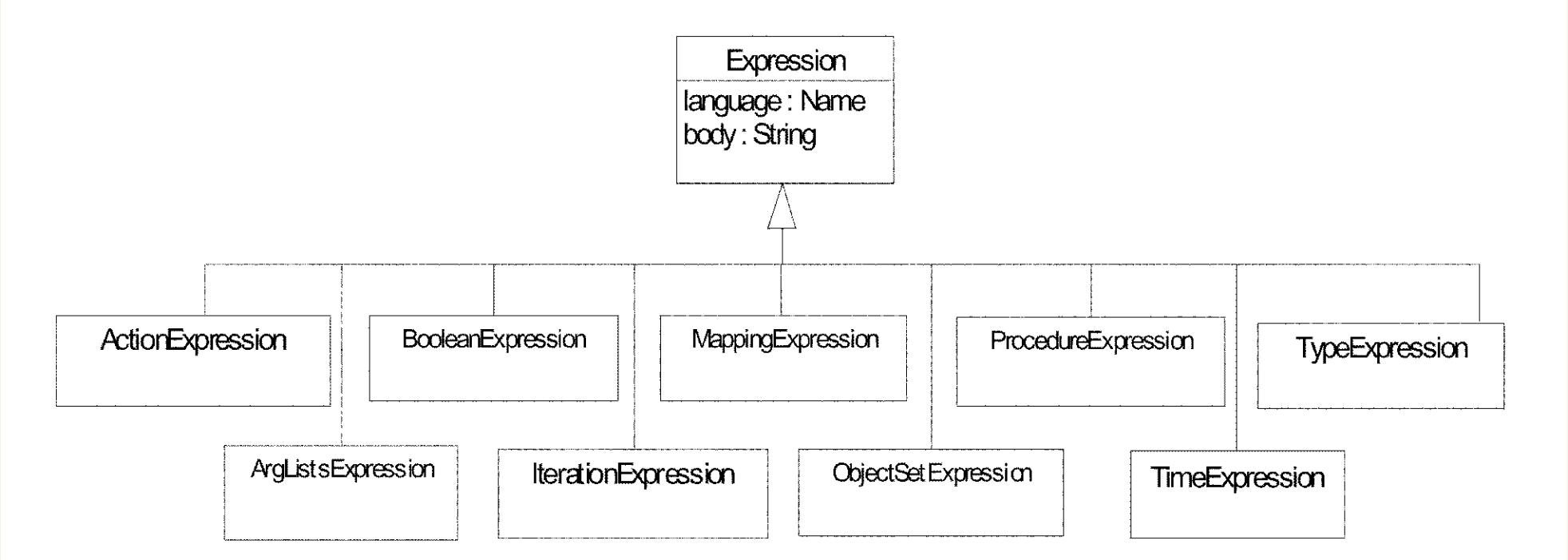


Figure 2-13 Behavioral Elements Package

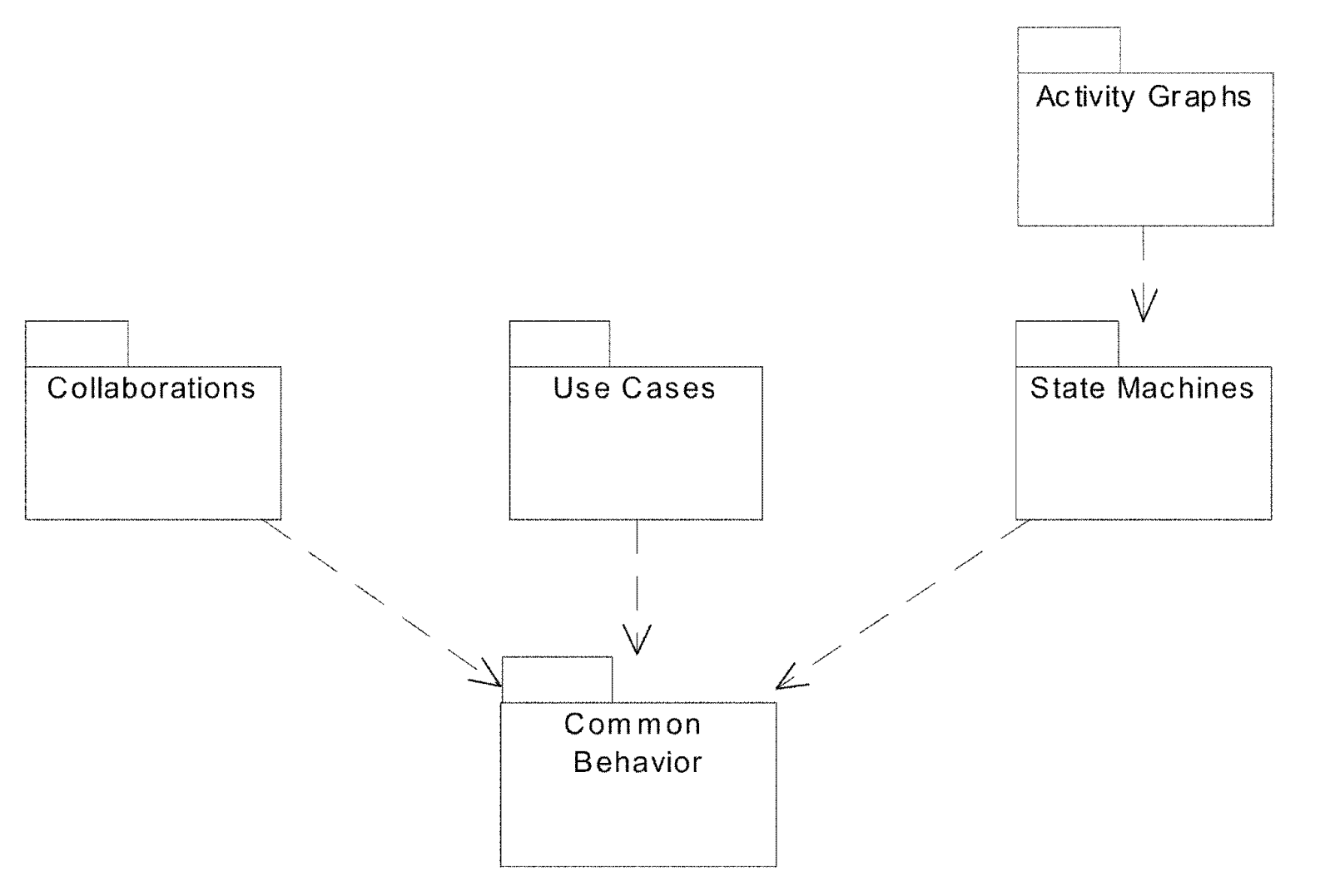


Figure 2-14 Common Behavior - Signals

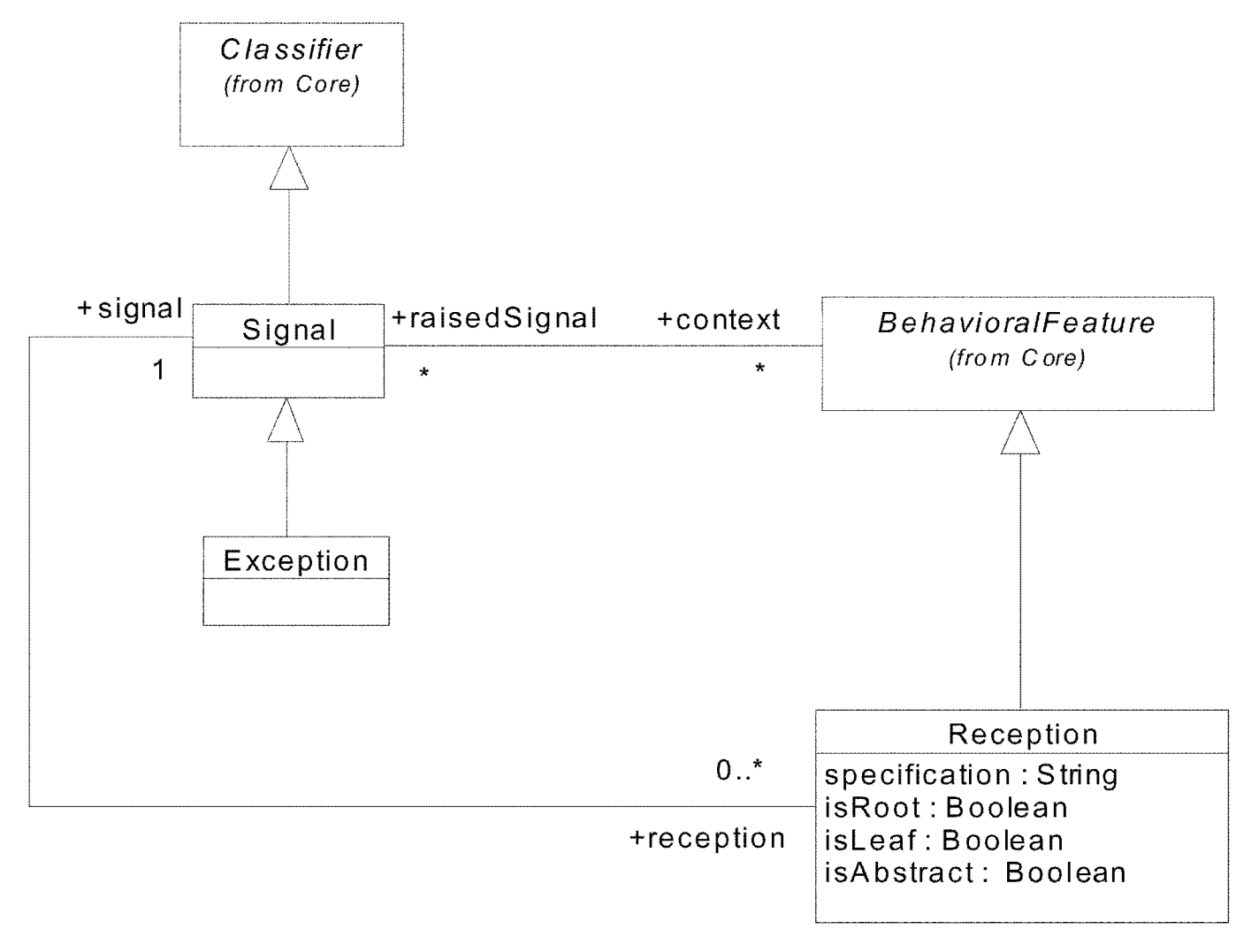


Figure 2-15 Common Behavior - Actions

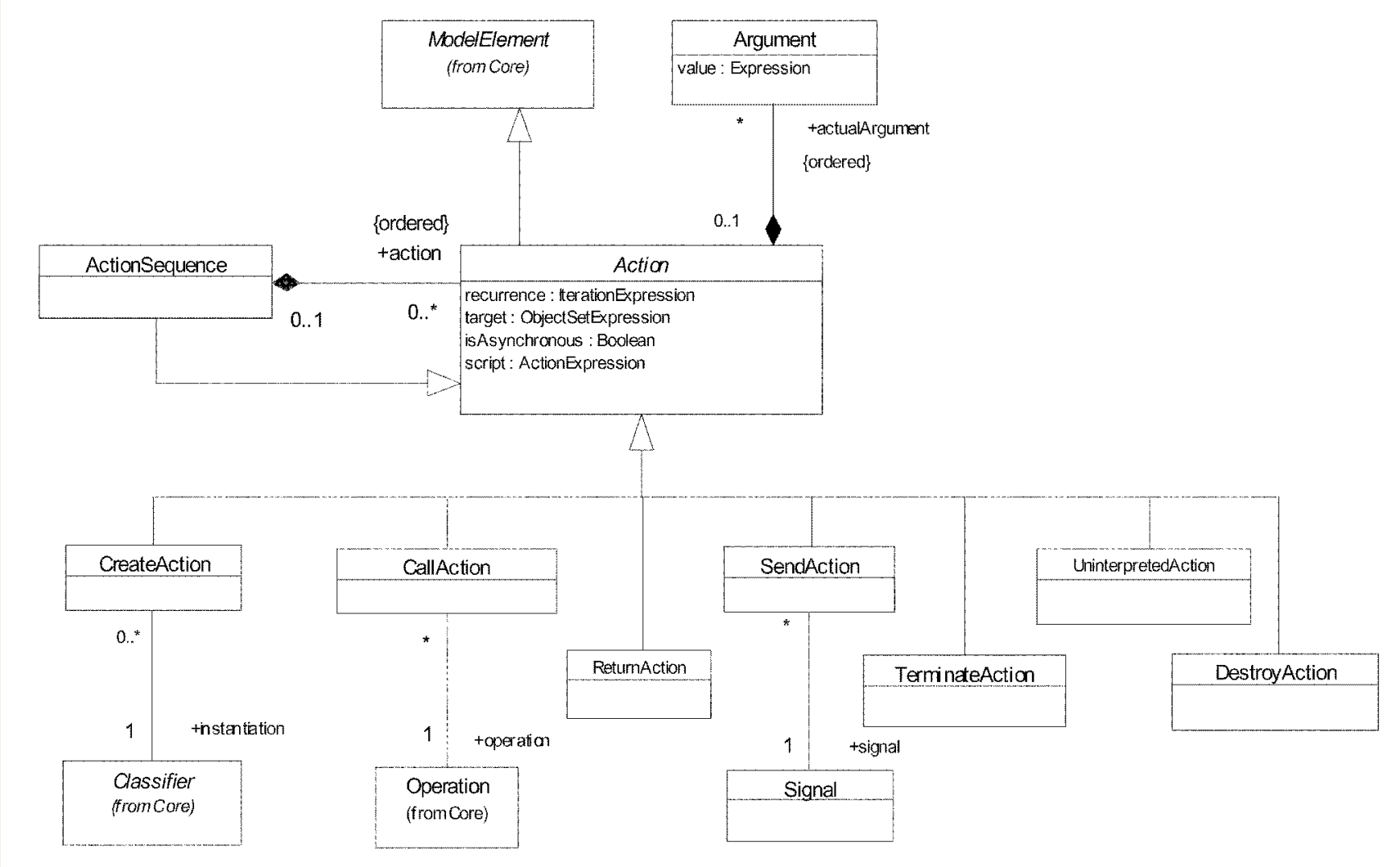


Figure 2-16 Common Behavior - Instances

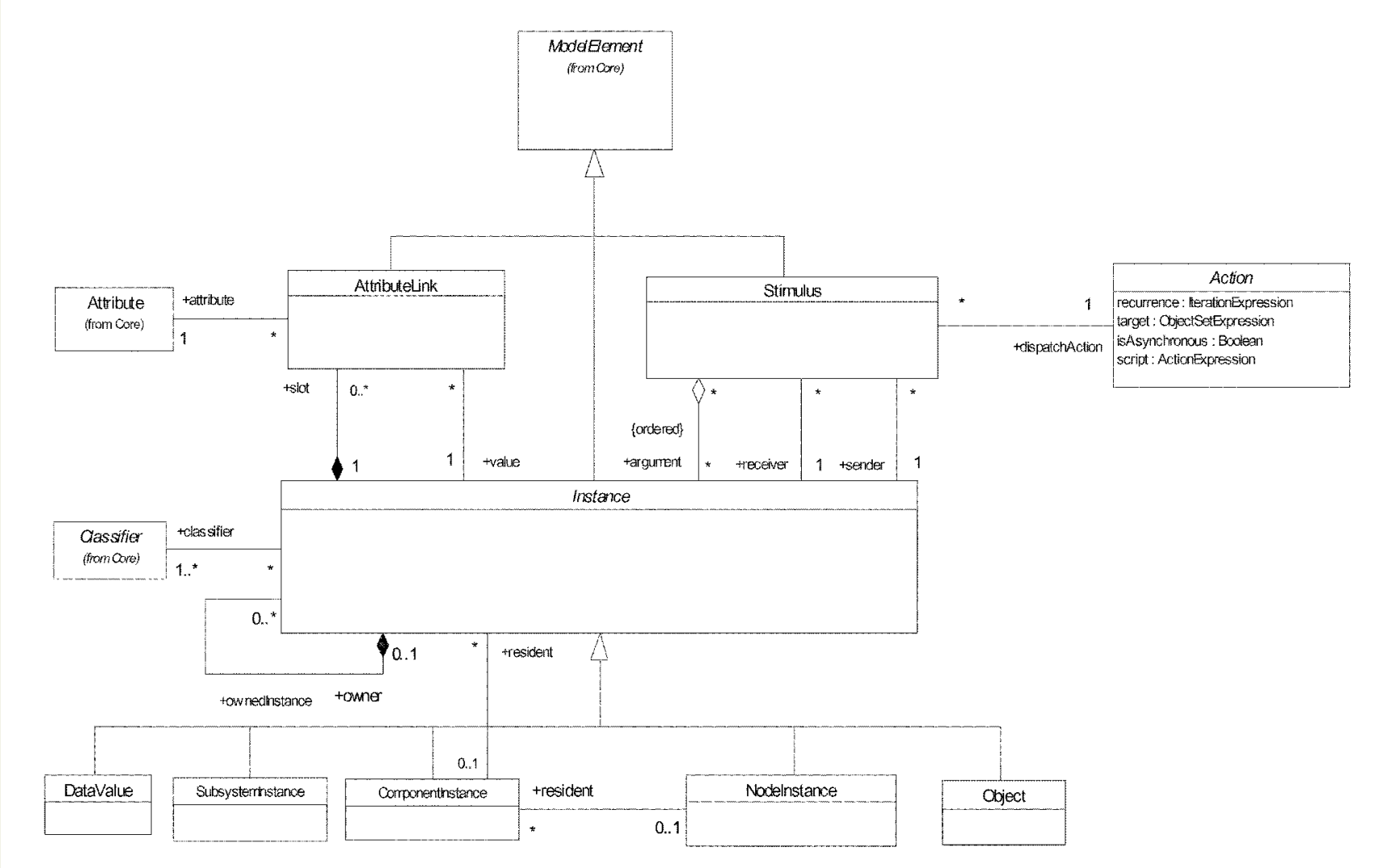


Figure 2-17 Common Behavior - Links

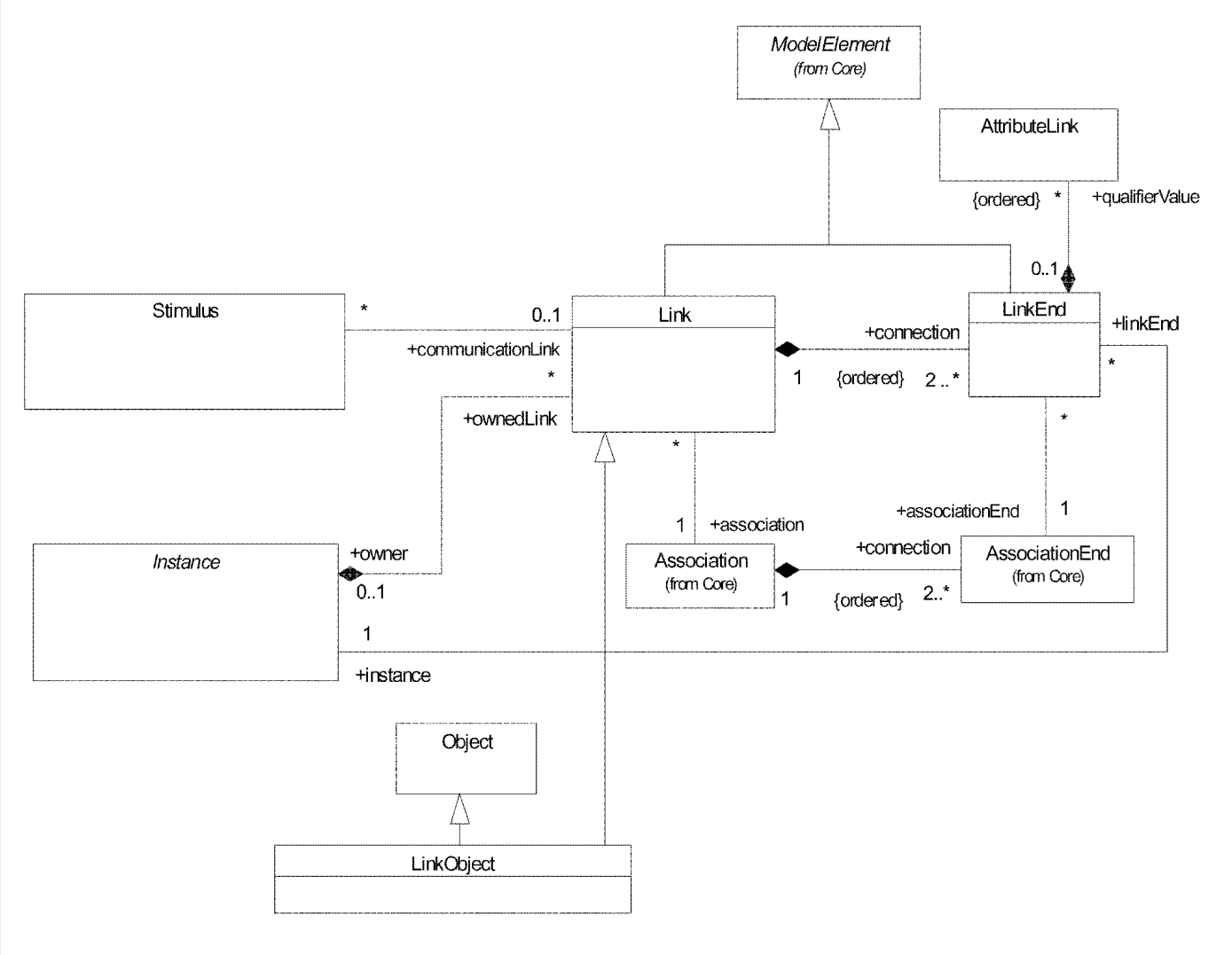


Figure 2-18 Collaborations - Roles

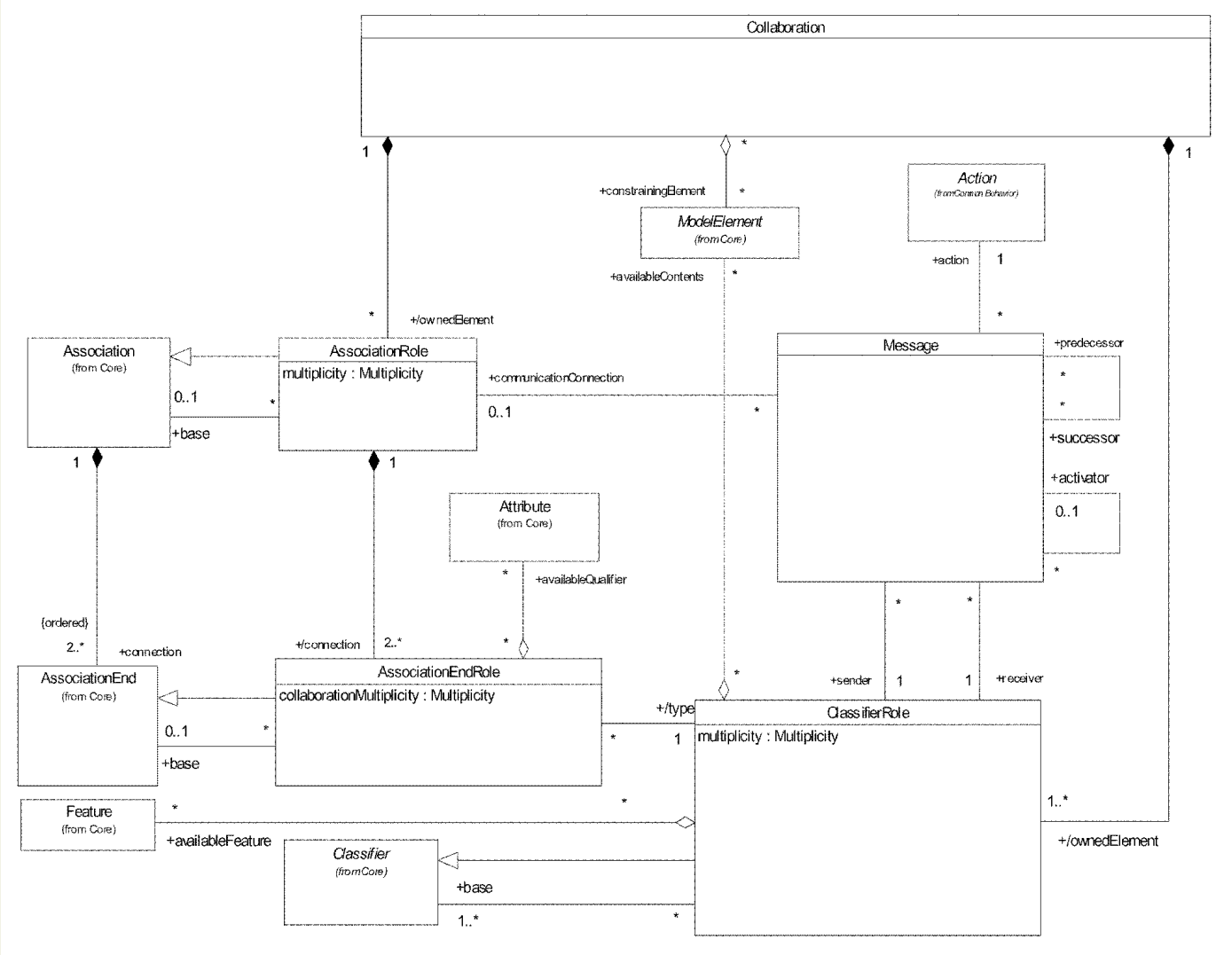


Figure 2-19 Collaborations - Interactions

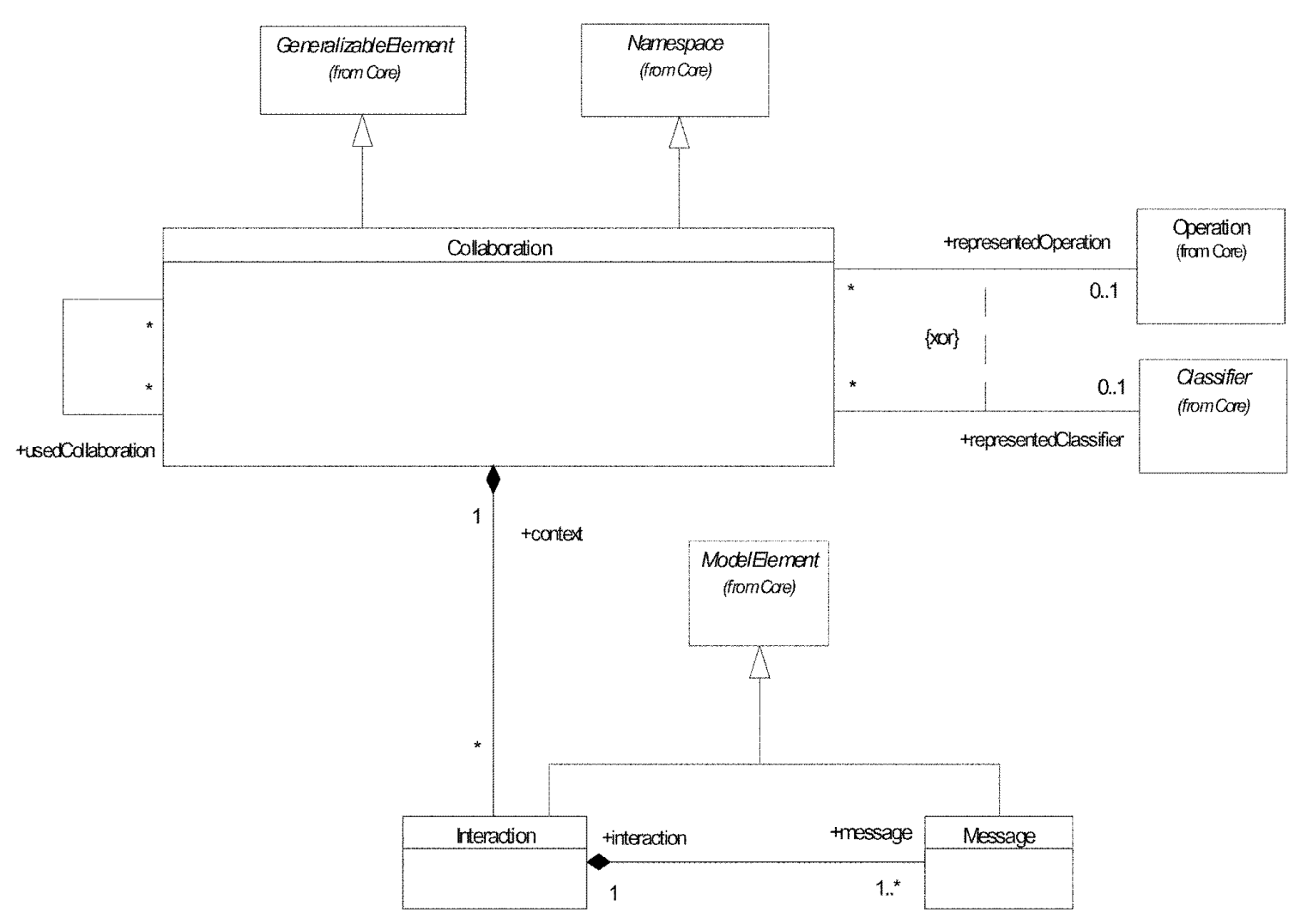


Figure 2-20 Collaborations - Instances

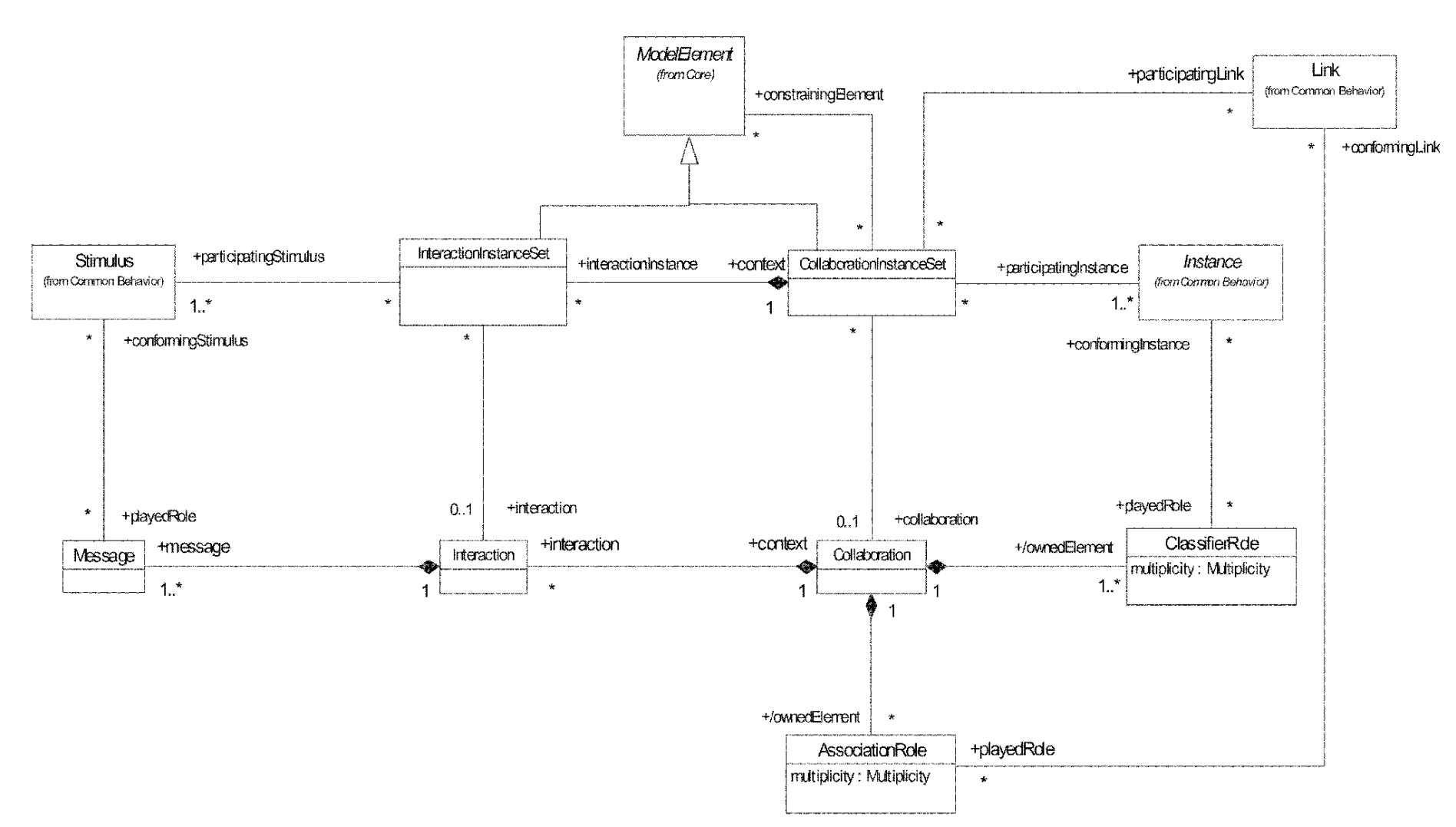


Figure 2-21 Use Cases

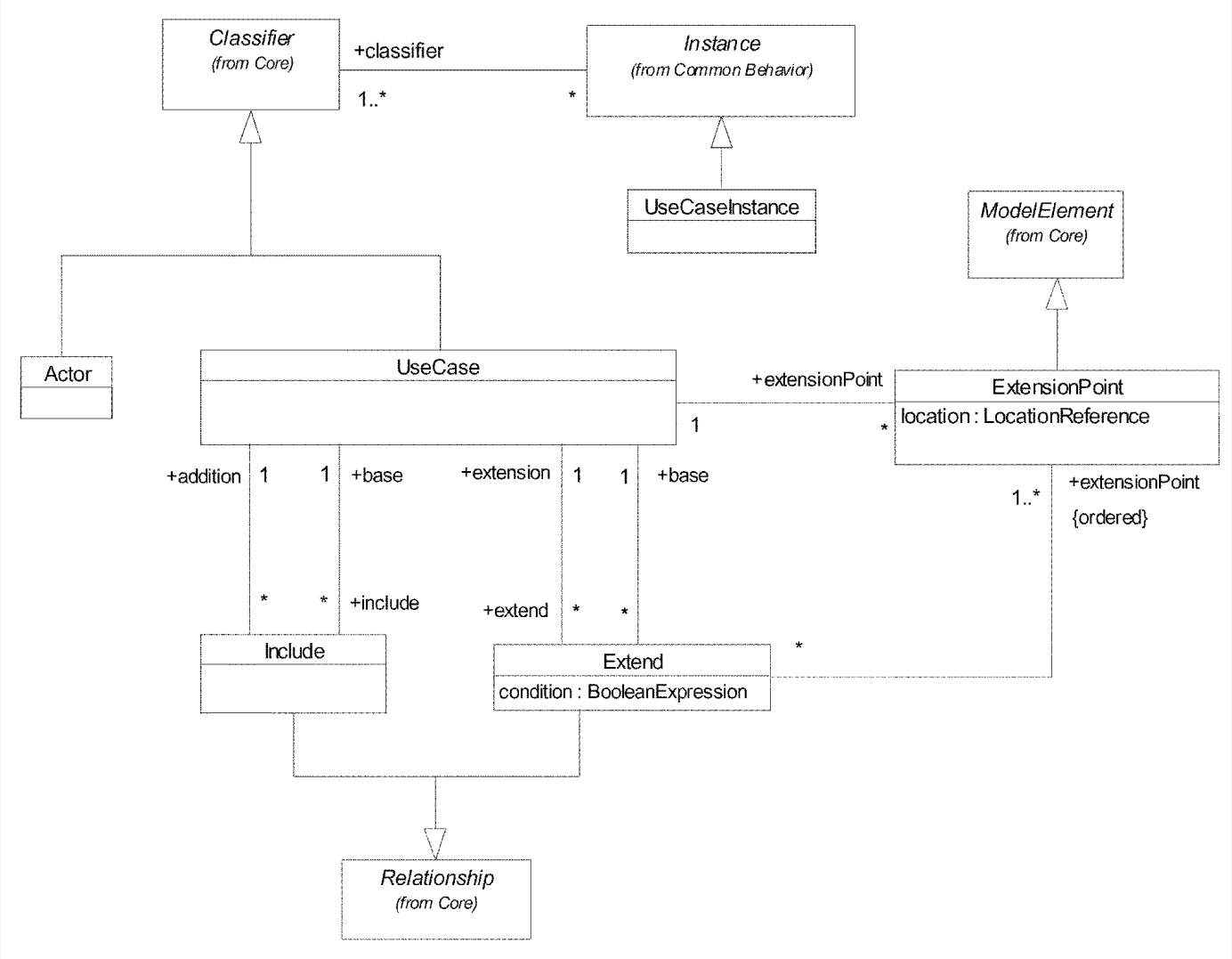


Figure 2-24 State Machines - Main

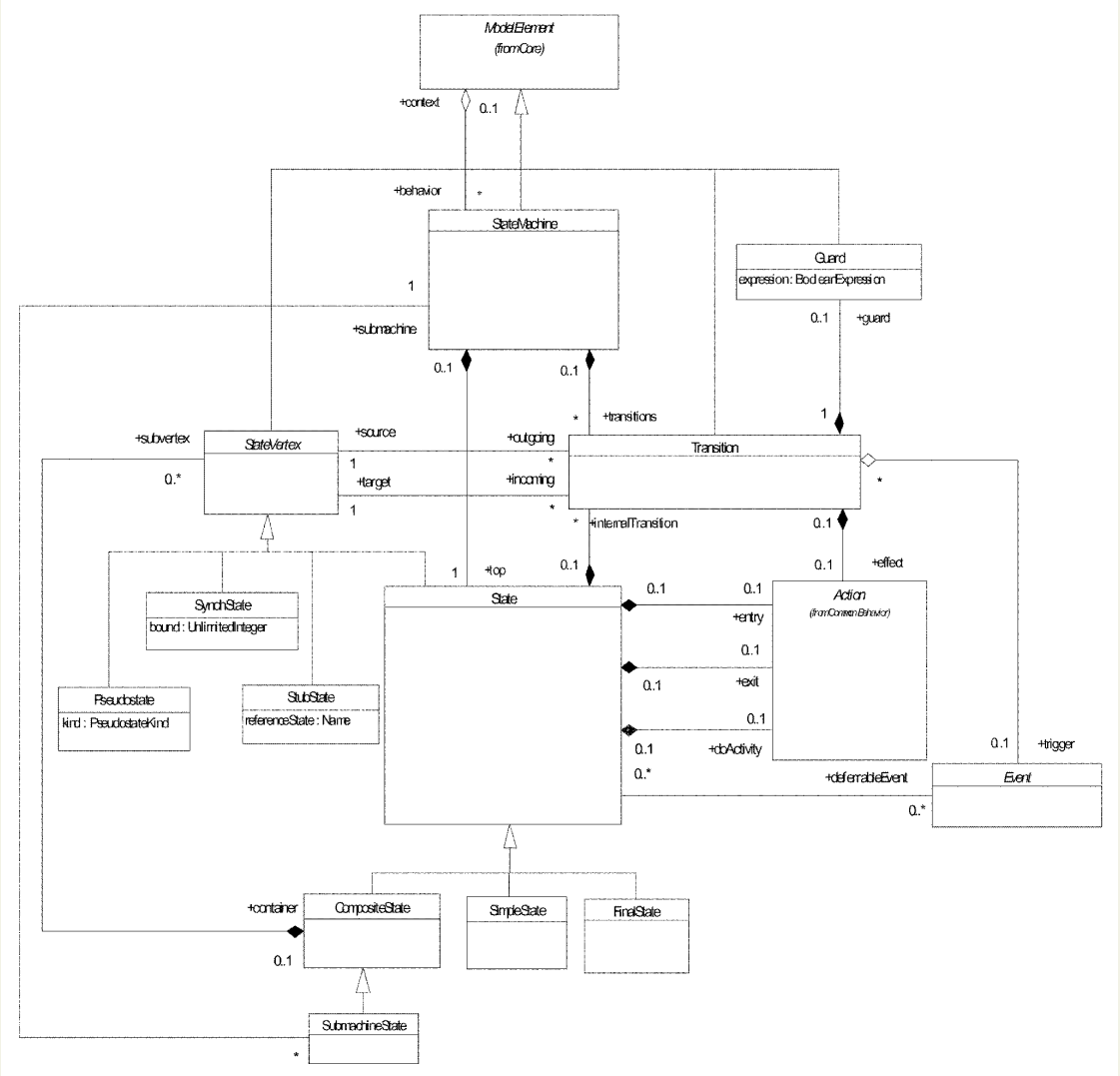


Figure 2-25 State Machines - Events

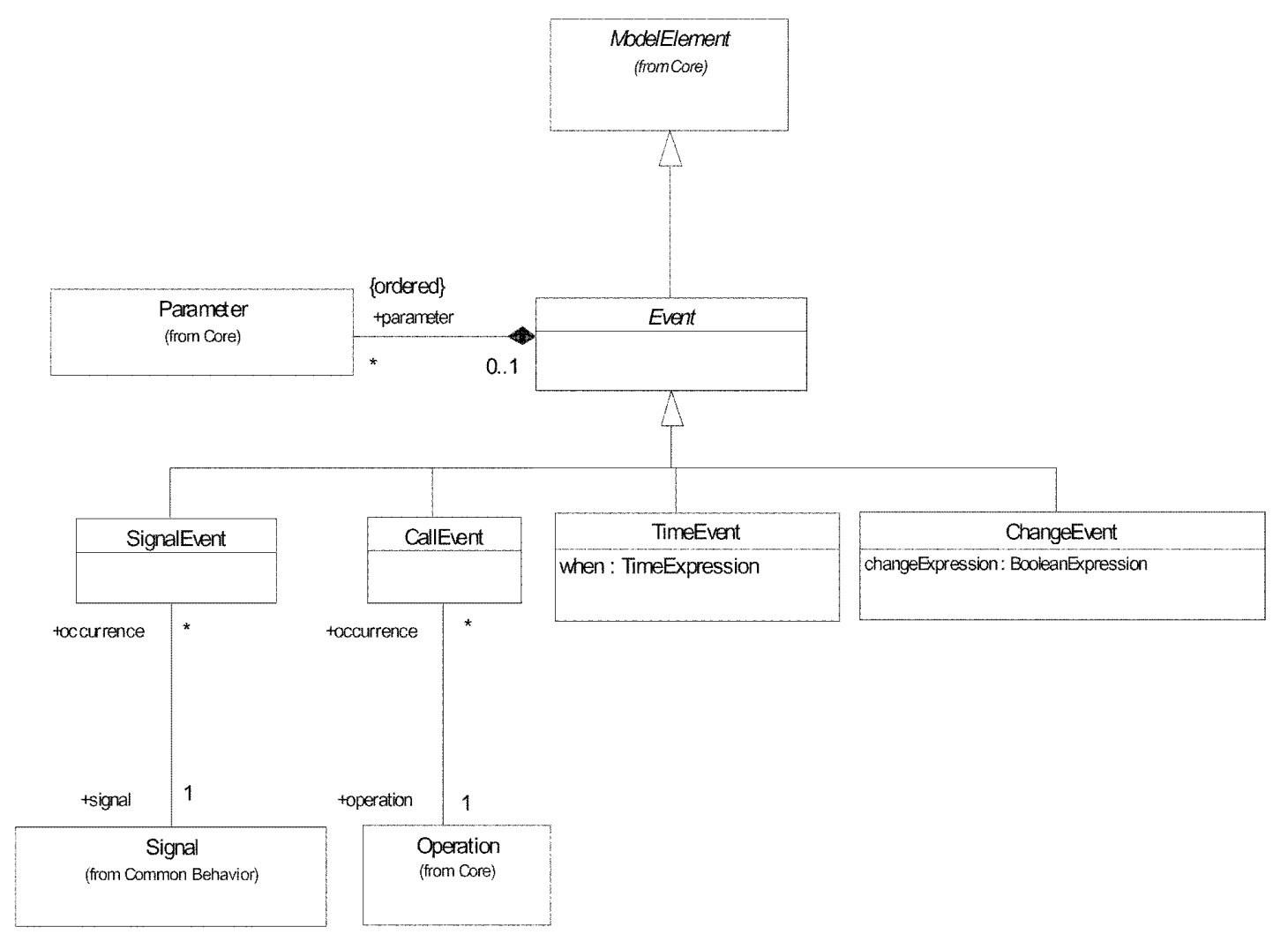


Figure 2-30 Activity Graphs

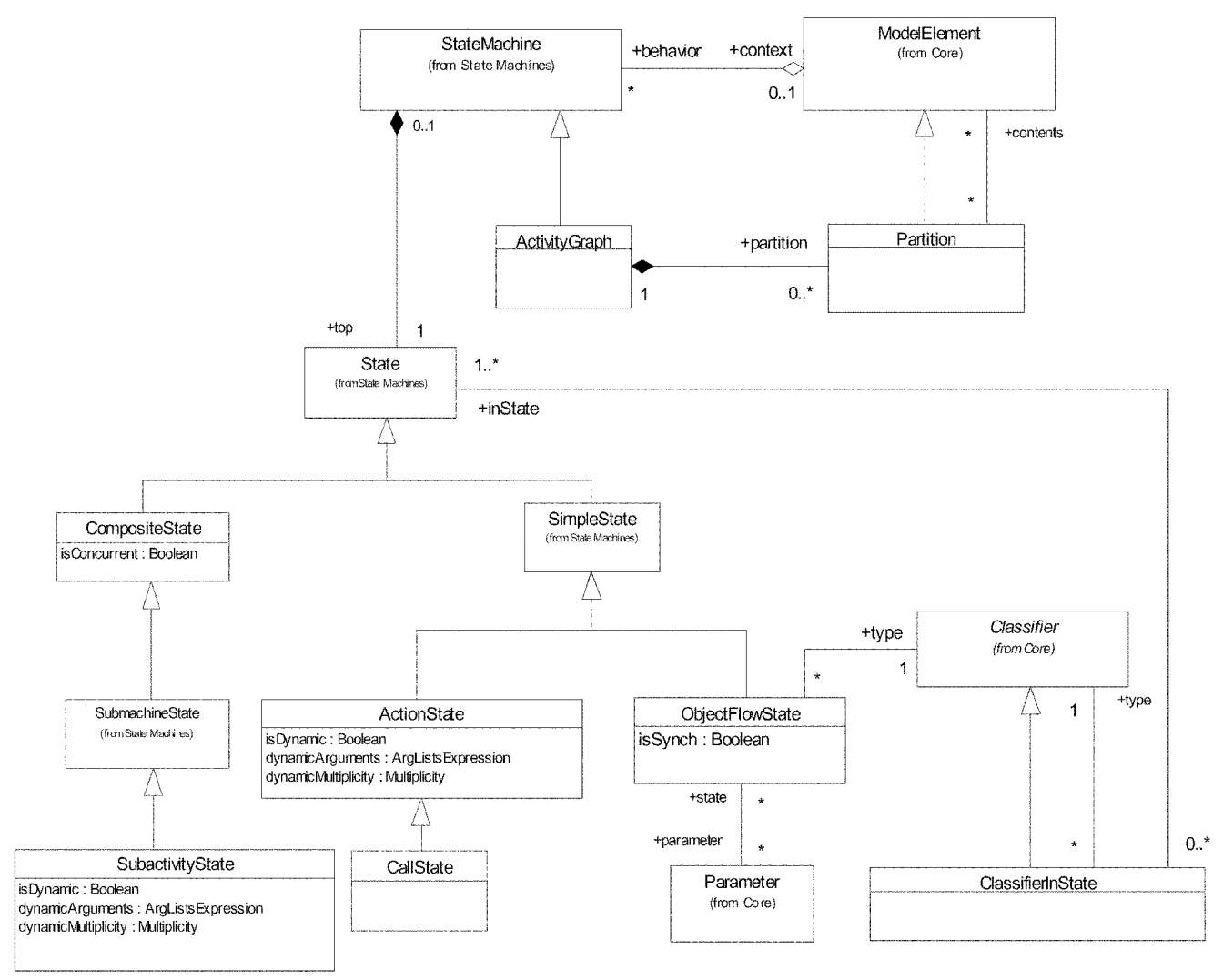


Figure 2-32 Model Management

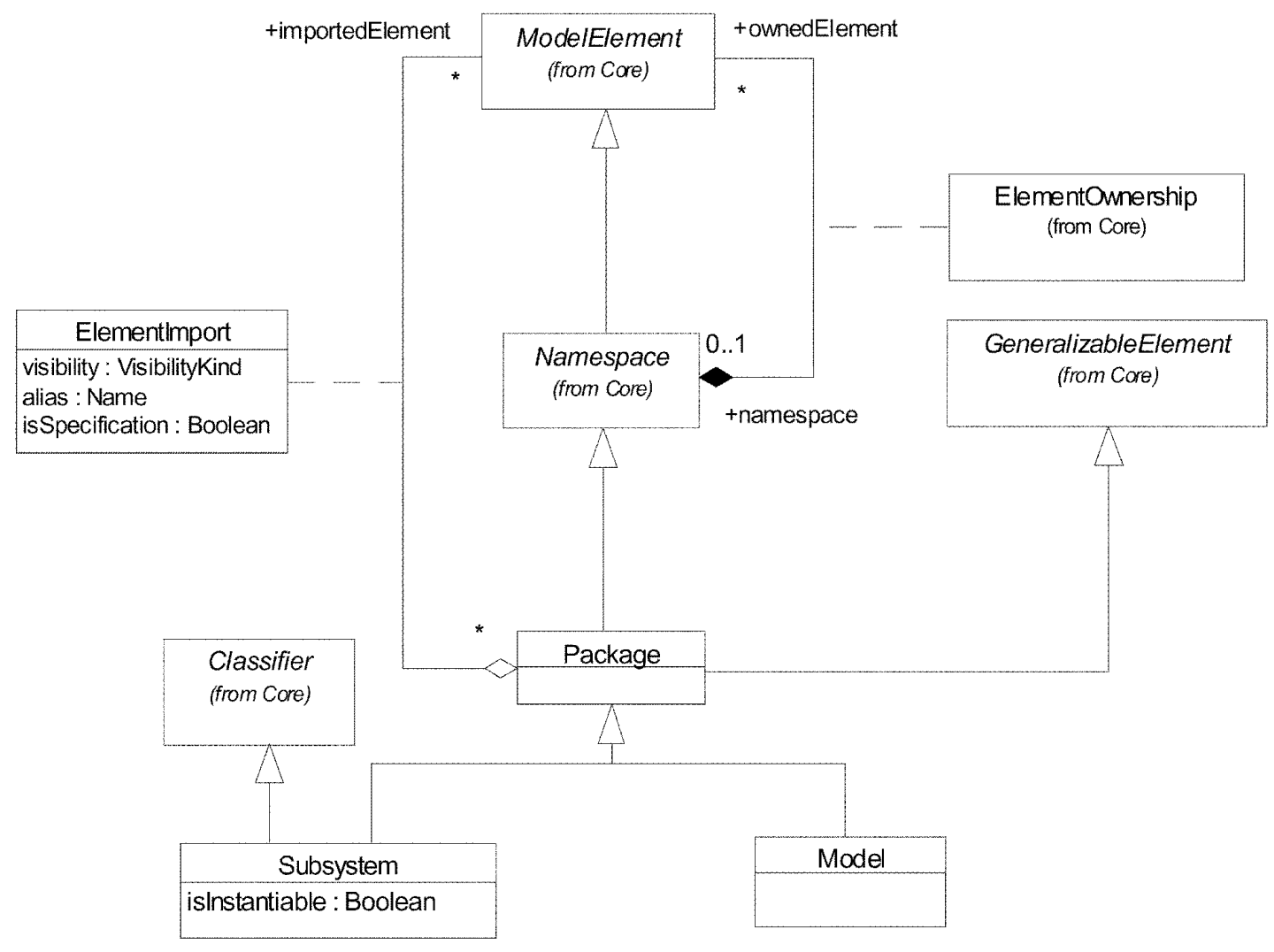


Figure 2-34 Subsystem illustration - shows Subsystem and its environment in the metamodel by flattening the inheritance hierarchy.

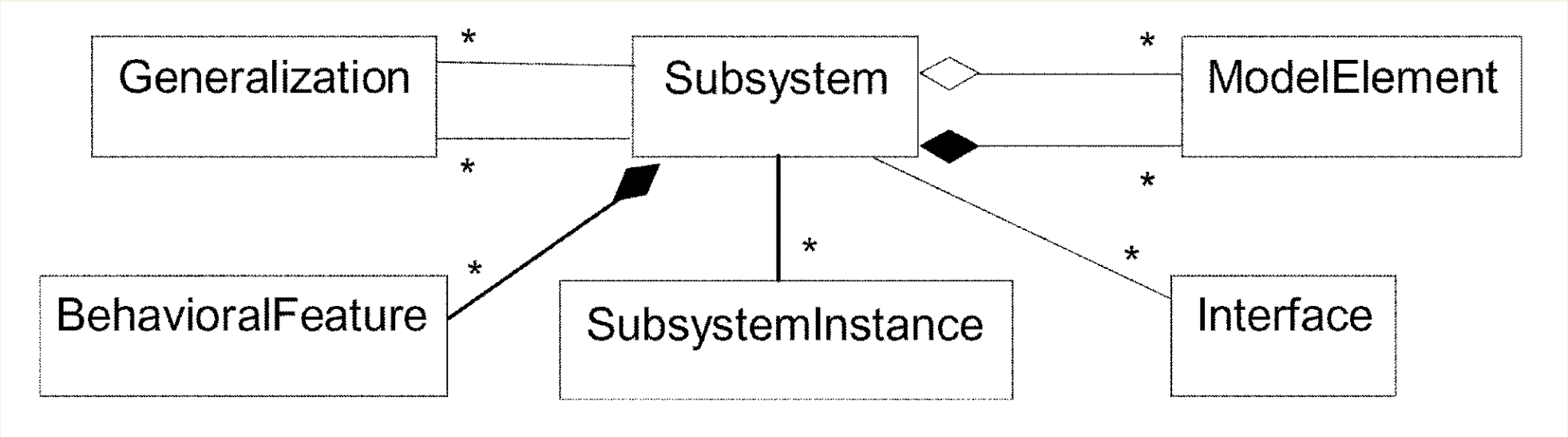


Figure 2-35 Model illustration - shows Model and its environment in the metamodel by flattening the inheritance hierarchy.