# Ch. 5.1: Part-B Advanced Structural Specification in UML

H.S. Sarjoughian

CSE 460: Software Analysis and Design

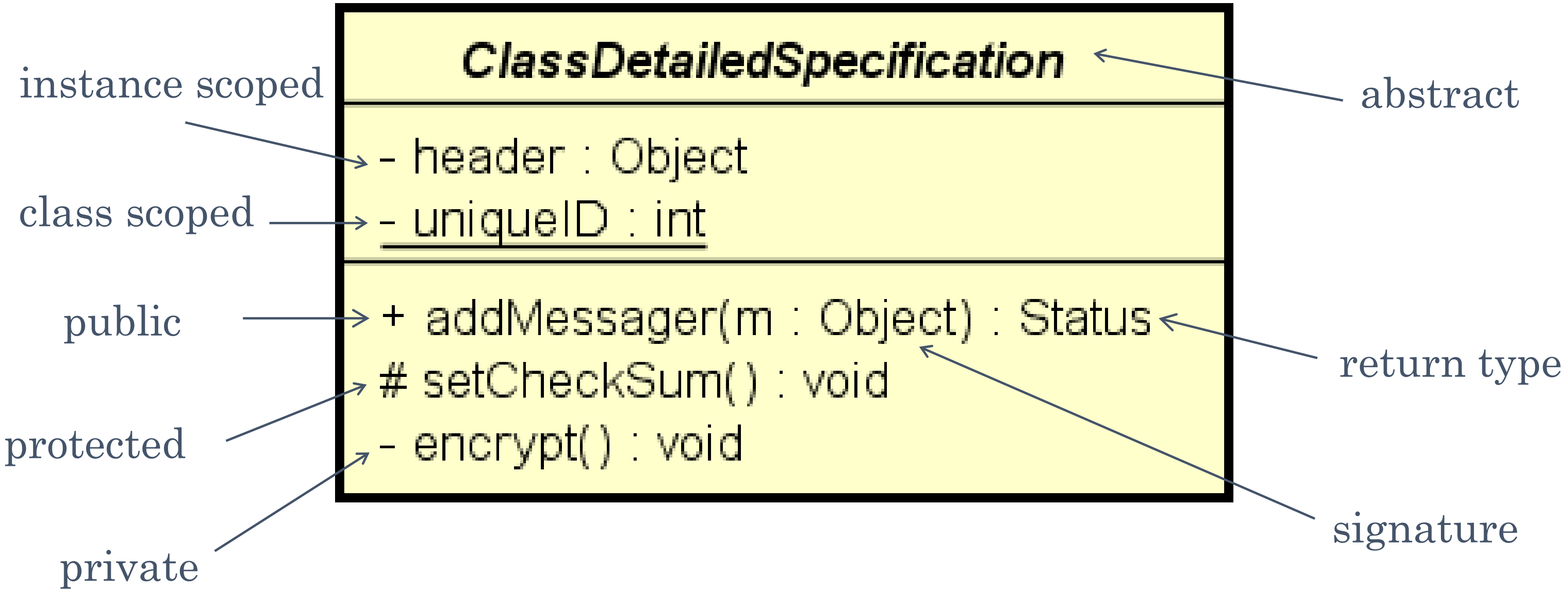
School of Computing, Informatics and Decision Systems Engineering
Fulton Schools of Engineering

Arizona State University, Tempe, AZ, USA

# *Advanced Properties of a Class*

UML allows specifying a class at any level of details. E.g.,

- Attributes: scope, type, …
- Operations: visibility, parameters, return type, …



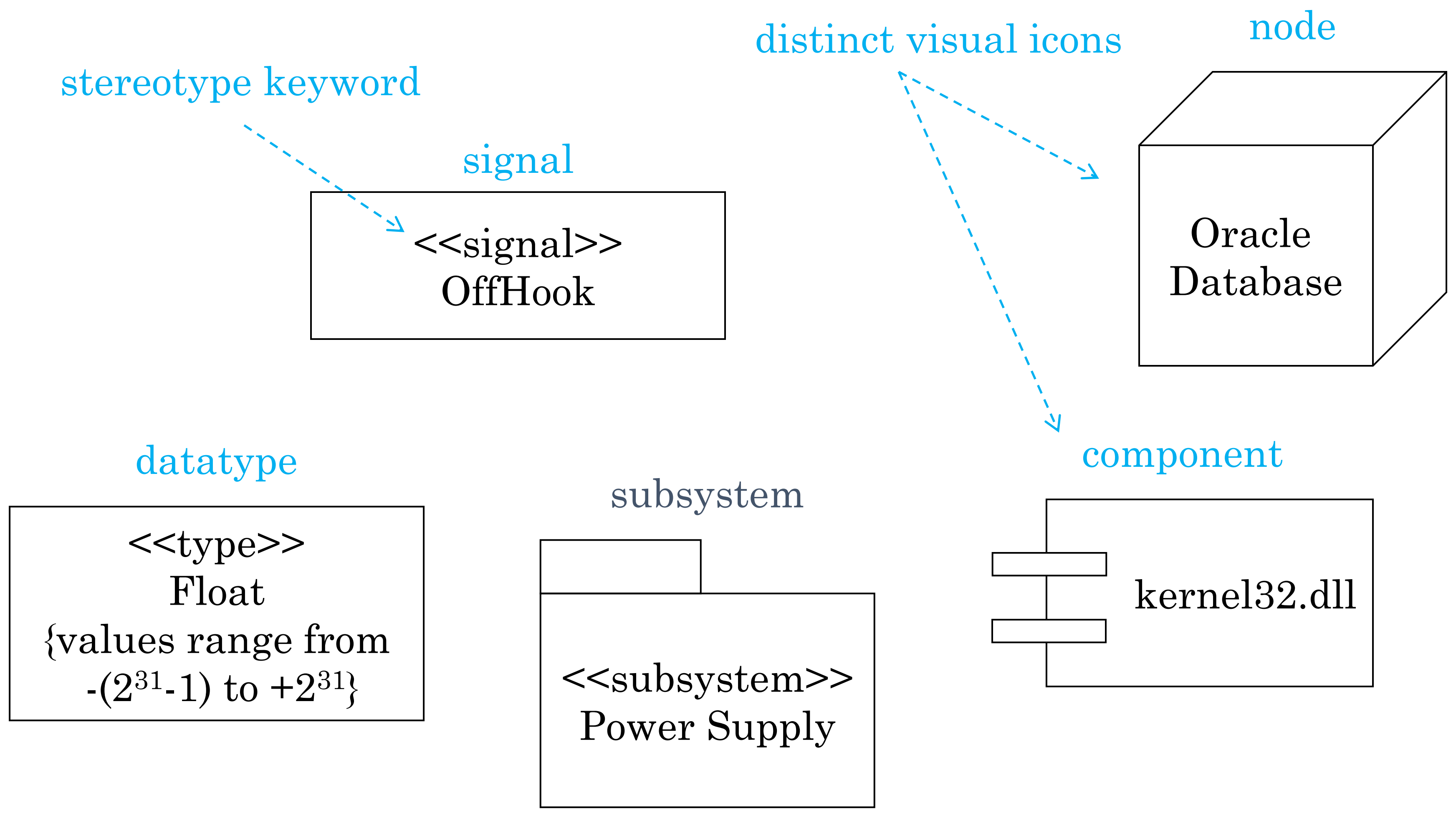
## *Classifiers*

A classifier represents a discrete modeling concept having identity, state, behavior, and relationships. Eight of these classifiers are:

- Datatype: a type whose values have no identity (primitive and enumeration types)
- Signal: asynchronous stimulus communicated between instances
- Component: a physical or replaceable part of a system that conforms to and provides the realization of a set of interfaces
- Node: a physical element that exists at run-time and that represents a computational resource
- Subsystem: a group of elements of which some constitute a specification of the behavior offered by the other contained elements
- Interface: a collection of operations that are used to specify a service of a class or a component
- Use-case: a description of a set of a sequence of actions, including variants, that a system performs to yield an observable result of value to an actor
- Class: a description of a set of objects that share the same attributes, operations, relationships, and semantics

some classifiers may have instances where every instance shares the same structural and behavioral features.

# Classifiers Graphical Notations



## *Visibility*

Visibility: specifies whether a class or more generally a classifier can be used by other classifiers; attributes and operations also have visibilities. Types of visibilities are:

- **public (+):** any classifier can use or access the feature
- **protected (#):** any descendent (child) can use or access the feature
- **private (-):** the classifier alone can use the feature

Notes: generally it is a good practice to make as little as possible fully visible (i.e., declared as public). **Visibility is a kind of abstraction and necessary for developing robust systems through appropriate use of information hiding**. It is often best to keep most if not all of a classifier's attributes private or protected. For operations only those that are to be made accessible to external users should be made public while keeping all others either protected or private.

## Scope

A classifier's attributes and operations can have scopes. The scope of a feature specifies whether it appears in each instance of the classifier (instance scoped) or there exists only a single instance shared by all (class scoped). Types of scopes are:

- **instance:** each instance of the classifier has its own copy of a feature
- **classifier:** all instances of a classifier share the same same copy of a feature

Notes: Most features of the classifiers should be instance scoped. Classifier scoped feature is useful for private attributes that must be shared among a set of instances (e.g., unique IDs).

## *Multiplicity*

A **class** may be specified to have zero, one, or multiple instances.

- abstract class (no instances)
- class (many instances, default case)
  - utility class (no instances)
  - singleton class (one instance)

Attributes can have multiplicity as well. An attribute with multiplicity is specified as:

connectionport [2 ..*] : Port

optional

## Attributes

Attributes form structural specification of a class. Attributes can be specified as names alone (most abstract form). Additional details can be specified:

- **visibility**
- **scope**
- **multiplicity**
- **type**
- **initial value**
- **Changeability**
  - **Changeable:** no restrictions
  - **Addonly:** attributes with multiplicity > 1, additional values may be added, but once created, it may not be removed or modified
  - **Frozen:** attribute’s value cannot be changed after initialization

```
[visibility] name [multiplicity] [: type] [= initial-value]
  [{property-string}]

    name [0 ..n] : String

    origin : point = (0, 0, 0)
```

## Operations

Operations form behavioral specification of a class. An operation can be specified as a name alone (most abstract form). Additional details can be specified:

- visibility
- scope
- parameter listing
- return type
- concurrency
- changeability
- stereotype

```
[<<stereotype>>] [visibility] name ([parameter-list]) [:
  return-type] [{property-string}]

    # set (n : name, s : String)

    getID( ) : Integer
```

## Operations (cont.)

**Signature (parameters)**

[direction] name : return-type [= default-value]

- **in:** an input parameter which may not be modified
- **out:** an output parameter; there is no input value;  the final value is available to the caller
- **inout:** an input parameter which may be modified
- **return:** a return value of a call

```
Matrix::transform (in distance: vector, in angle: real=0): return Matrix
```

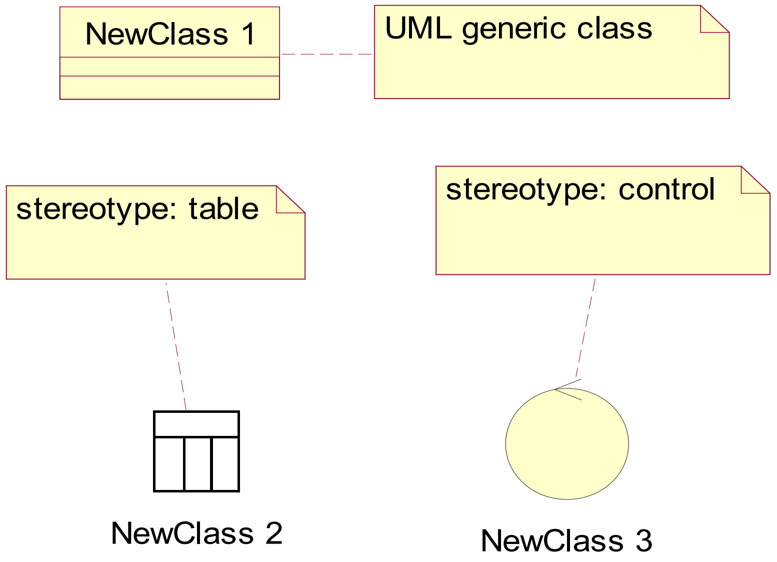
**Other properties**

- **leaf:** operation is not polymorphic
- **isQuery:** execution of the operation does not cause state change
- **sequential**
- **guarded**
- **concurrent**

} concurrency semantics

# Extensibility and Stereotypes

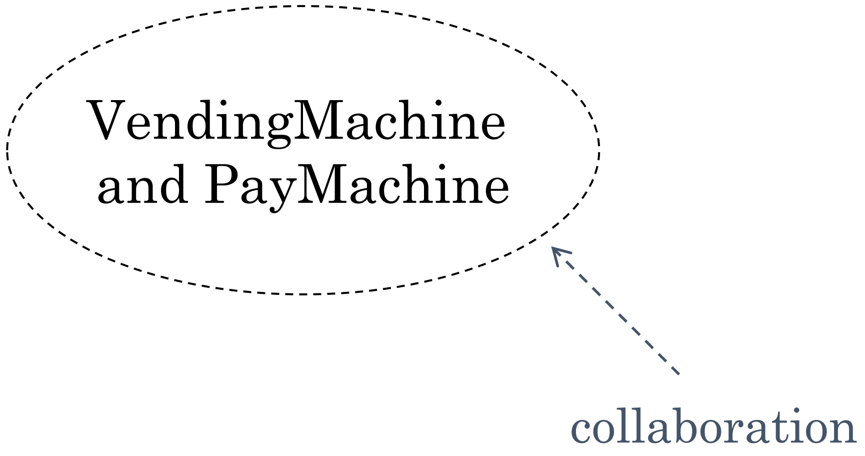
Sometime there is a need to model non-software elements (e.g., a workcell in a manufacturing line). Such elements can be modeled as classes which are adorned with stereotypes. These stereotype classes are distinguishable from native UML elements via their visual cues (icons).



UML modeling language can be extended via stereotypes and therefore new building blocks may be introduced

## Collaborations

- A collaboration is used as a representation of a society of classes, interfaces, and other elements that "collaborate" with one another to provide some cooperative behavior that is bigger than the sum of all parts.



## Some Stereotypes for Classes

UML defines a set of standard stereotypes for classes:

keyword

- utility: set of all attributes and operations are class-scoped
- interface: a collection of operations that are used to specify a service of a class or component
- implementationClass: the implementation of a class in some programming language
- exception: an event that may be thrown or caught by an operation
- process: a classifier whose instances represent a heavyweight flow of control
- signal: an asynchronous stimulus communicated among instances
- thread: a classifier whose instances represent a lightweight flow of control
- actor: a set of roles that users of use-cases play when interacting with these use-cases.

## Classifier Specification Levels

UML supports from informal to formal specifications

- Specify the responsibility of class (obligation or contract) in a note attached to the class (least formal)
- Specify the semantics of the class as a whole using text – rendered in a note attached to the class
- Specify the body of each method using text or programming language – rendered in a note attached to the operation by a dependency relationship

## *Classifier Specification Levels (cont.)*

- Specify pre- and post-conditions for each operation as well as the invariants of the class as a whole using text – rendered in a note attached to the class
- Specify a state machine for the class
- Specify a collaboration that represents the class
- Specify pre- and post-conditions as well as the invariants using a formal language such as object constraint language (most formal)

## *Hints on Developing Classifiers*

A **well-structured** classifier (why?)

- **contains both structural & behavioral aspects**
- **is well balanced – tightly cohesive and loosely coupled**
- **exposes only what is necessary for its intended clients to use it**
- **is unambiguous in its intent/semantics**
- **is specified in such a way it does not eliminate all degrees of freedom or renders its meaning ambiguous**

**Useful visual representation** (why?)

- shows only the properties that are important in aiding its understanding
- uses good stereotypes to provide visual cues for the intent of the classifier

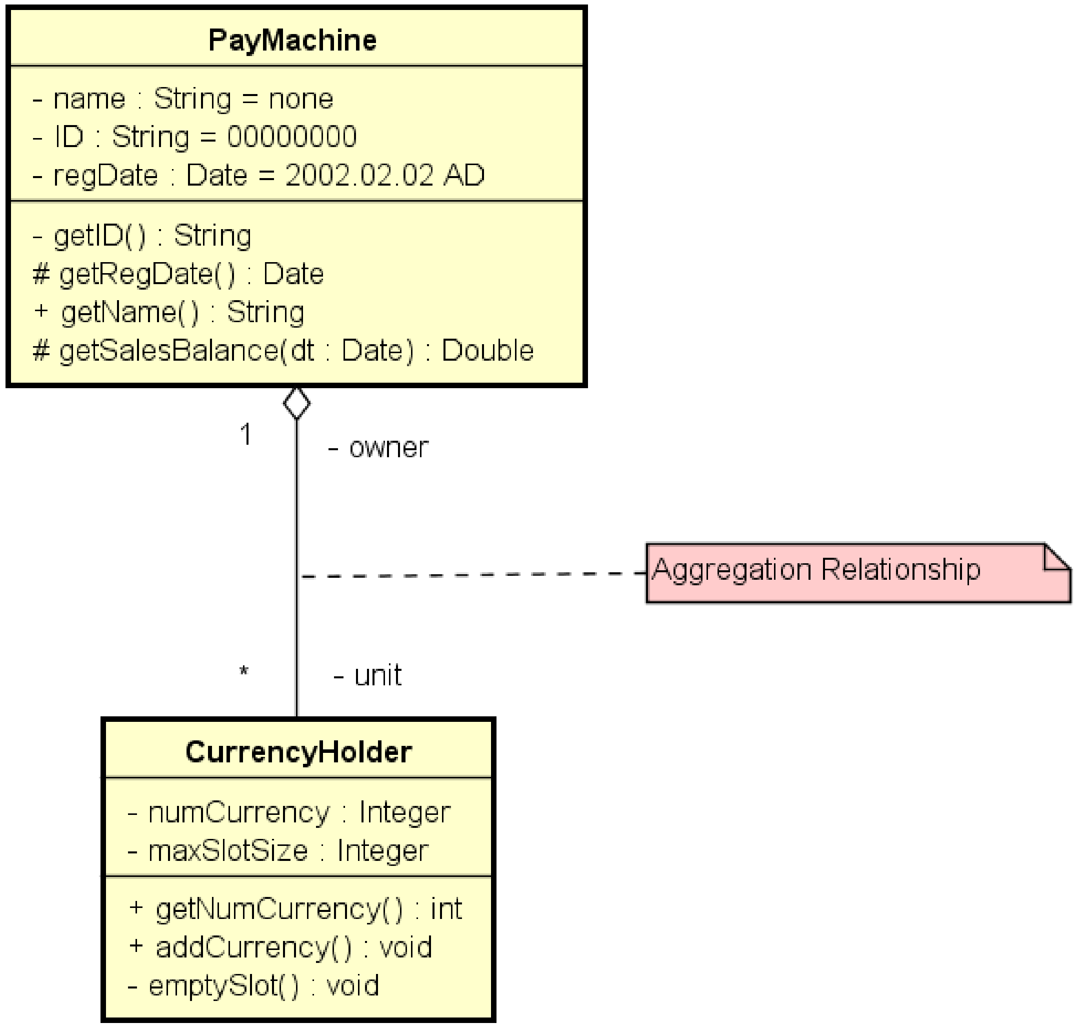
## Advanced Features for Common Relationships

UML offers five relational building blocks. These relationships (e.g., dependency and realization) are most used and most important in a variety of diagrams including class and object diagrams.

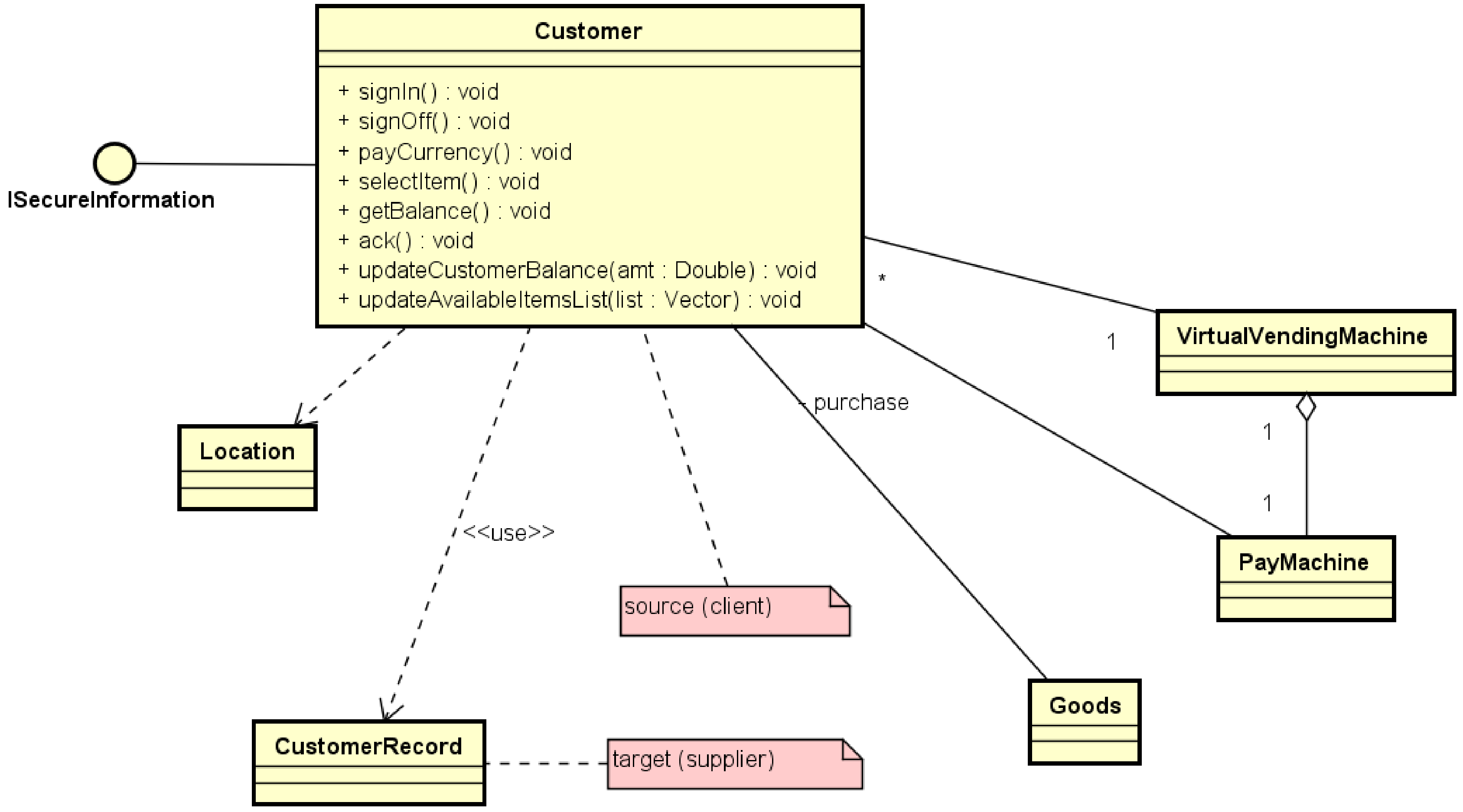
- Generalization
- Aggregation
- Dependency
- Realization
- Association

These relationships can specify a web of relationships. Advanced features can be used to provide detailed kinds of relationships.

# Advanced Class Diagram Example



# Advanced Class Diagram Example (cont.)



# Generalization Relationship

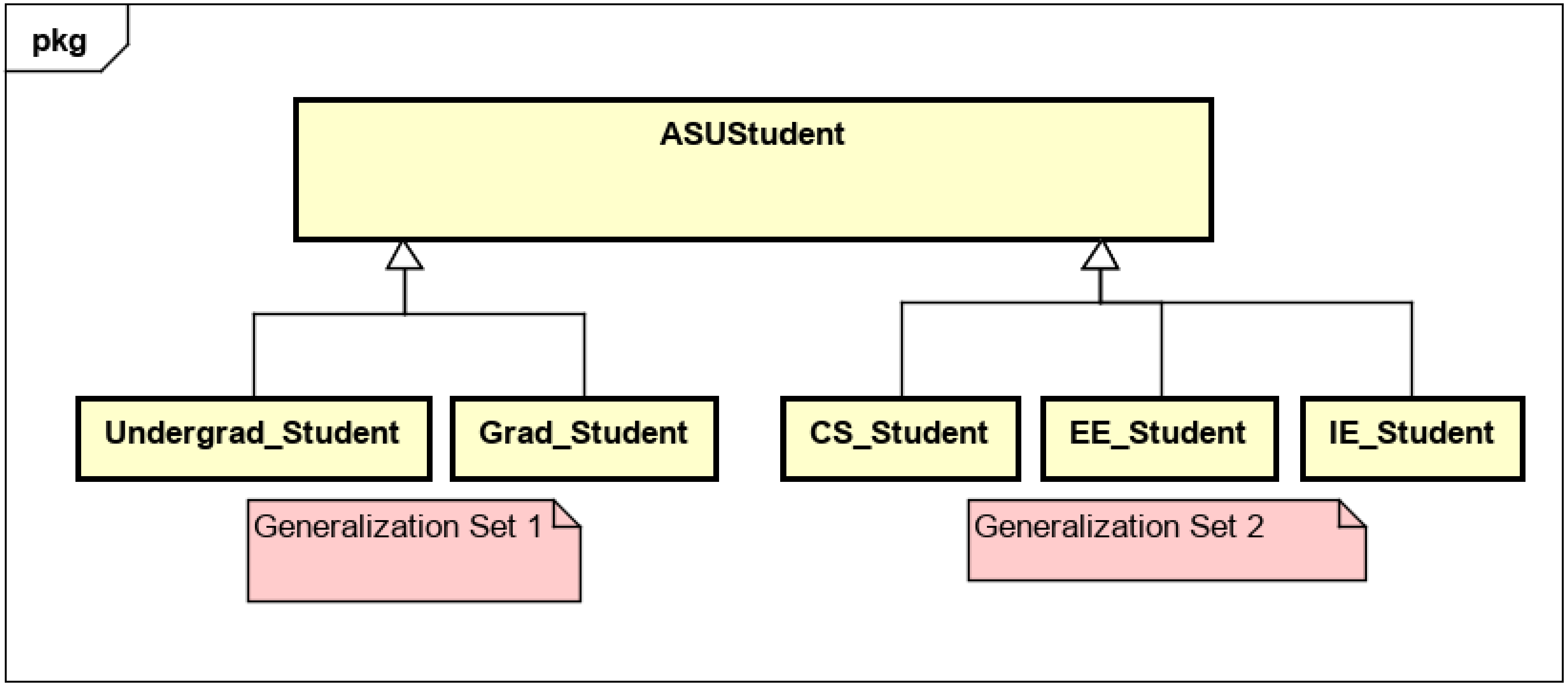
Generalization relationship holds between a **superclass/parent** thing and **subclass/child** thing. Subclass inherits all structure and behavior from its superclass. Further, subclass may add its own new structure and behavior, or just modify the behavior of its superclass. UML offers one stereotype:

- **implementation:** specifies that the child inherits the implementation of the parent but does not make public nor support its interface, thereby violating substitutability.

Constraints that can be applied to the generalization relationship are:

- **complete:** specifies that no additional children are allowed
- **incomplete:** specifies that not all children have been specified – additional children are permitted
- **disjoint:** specified that objects of the parent may have no more than one of the children as a type – static classification
- **overlapping:** specifies that objects of the parent may have more than one of the children as a type – dynamic classification

## Generalization (Inheritance) Relationships

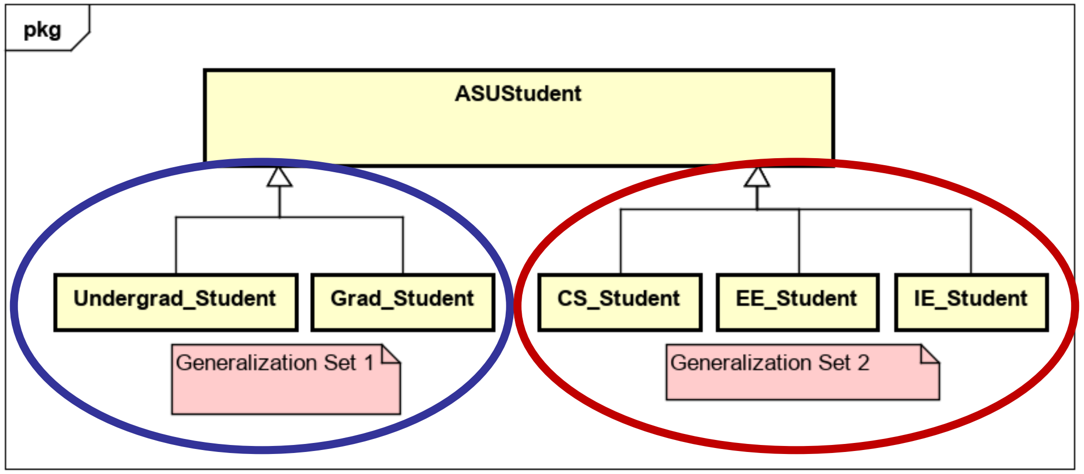


## Generalization Relationships

Constraints for generalization relationships

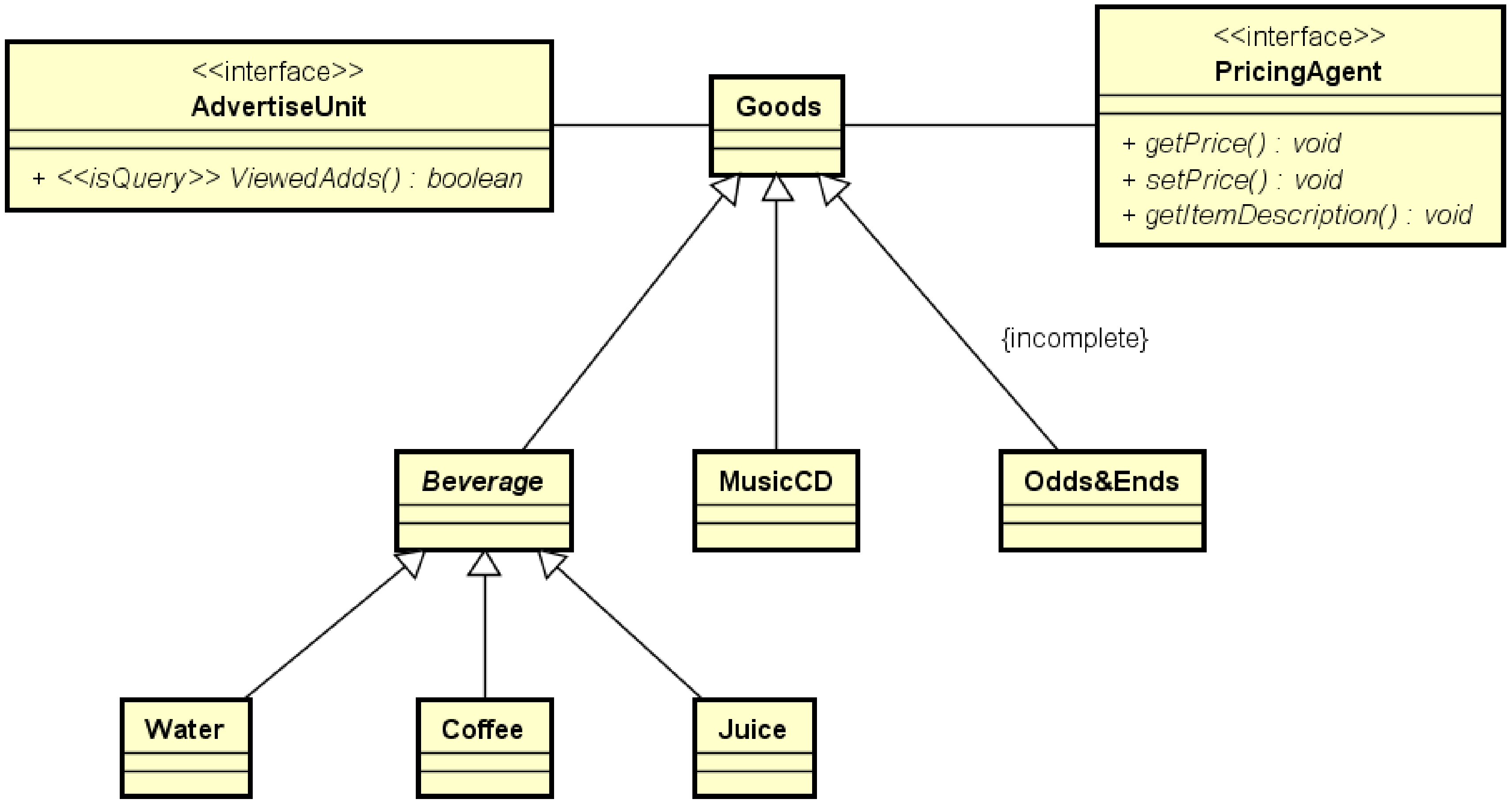
| | |
|---|---|
| {complete, disjoint} | Indicates the generalization set is covering and its specific Classifiers have no common instances. |
| {incomplete, disjoint} | Indicates the generalization set is not covering and its specific Classifiers have no common Instances. |
| {complete, overlapping} | Indicates the generalization set is covering and its specific Classifiers do share common instances. |
| {incomplete, overlapping} | Indicates the generalization set is not covering and its specific Classifiers do share common instances. |

## Generalization (Inheritance) Relationships



| | Complete | Incomplete | Disjoint | Overlapping |
|---|---|---|---|---|
| Set 1 | √ | | √ | |
| Set 2 | | | | |

# Advanced Class Diagram Example (cont.)



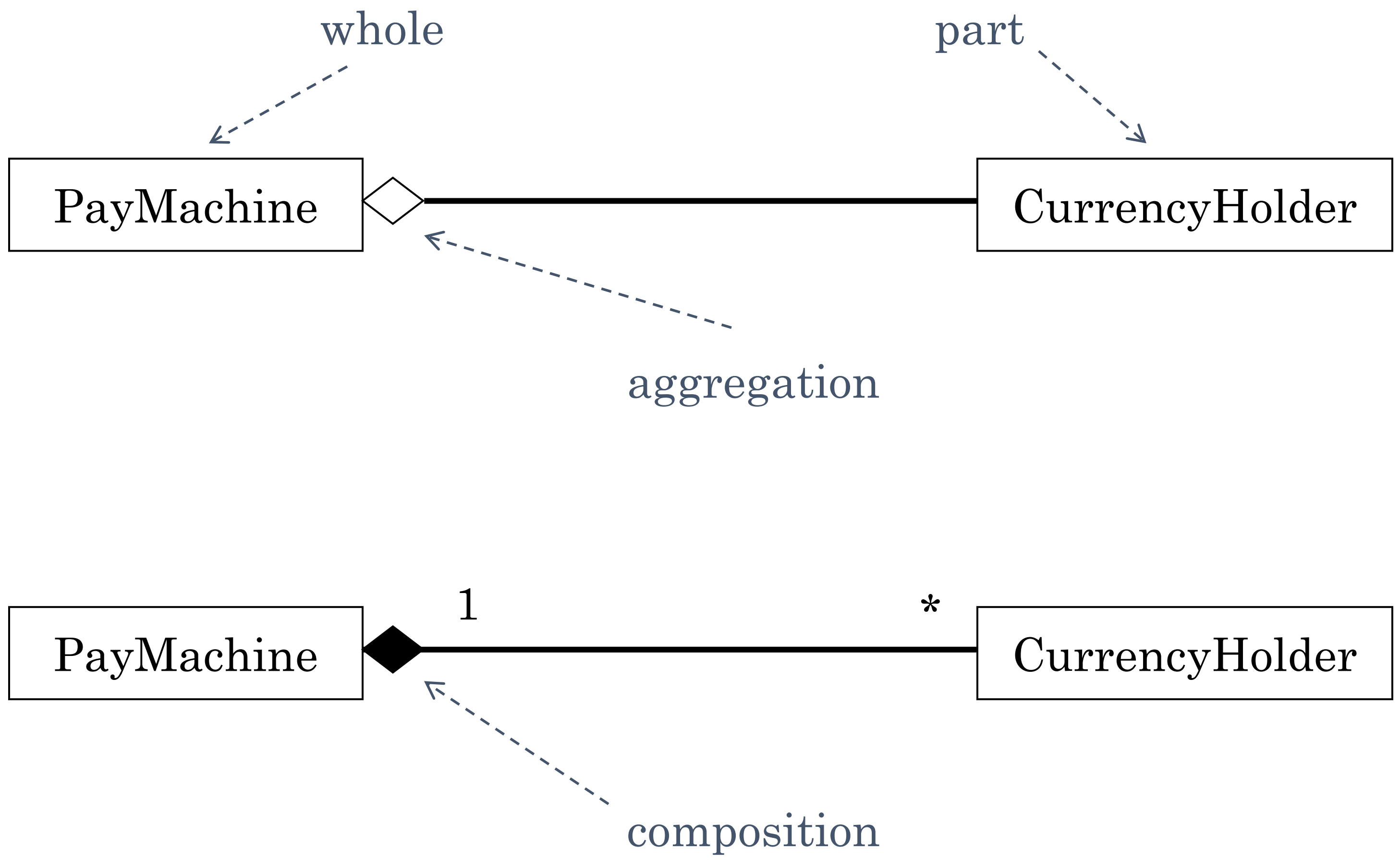
## *Aggregation and Composition Relationships*

Aggregation: specifies "**part**" from the "**whole**". This is a simple conceptual relationship – e.g., it does not specify how creation and lifetime of the whole and the part may be related to one another.

**Composition:** a variation of aggregation. In addition to distinguishing the part and the whole from one another, it specifies **ownership** and **coincident** lifetime of the part w.r.t. the whole:

- An object may be part of only one composite (whole) at a time
- Composite is responsible for creation and destruction of its parts
- Parts with varying multiplicity may be created after the composite itself, but once created, they live and die with it.

## *Aggregation and Composition Relationships (cont.)*



## Dependency Relationship Stereotypes

A dependency relationship (among classes and objects in class diagrams) specifies that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Some stereotypes that apply to the dependency relationships are:

- **friend:** specifies that the source is given special visibility into the target
- **instanceOf:** specifies that the source object is an instance of the target classifier
- **instantiate:** specifies that the source creates instances of the target
- **use:** specifies that the semantics of the source element depends on the semantics of the public part of the target

## *Dependency Relationship Stereotypes (cont.)*

Dependency relationships among **use-cases** may have two kinds of stereotypes:

- **include:** specifies that the source use-case explicitly incorporates the behavior of the target use-case
- **extend:** specifies that the target use-case extends the behavior of the source use-case

Dependency relationships among **objects** have three kinds of stereotypes:

- **become:** specifies that the target is the same object as the source but at a later time, possibly having its own values, state, or roles
- **call:** specifies that source operation invokes the target operation
- **copy:** specifies that the target object is an exact, but independent, copy of the source

# *Dependency Relationship Stereotypes (cont.)*

Stereotypes for dependency relationship for **state machines**

- **send:** specifies that source operation sends the target event

Stereotypes for dependency relationships among **packages**

- **access:** specifies that the source package is granted the right to reference the elements of the target package
- **import:** specifies that the public contents of the target package enter the flat namespace of the source

# *Modeling of Relationships*

## Software Engineering Process

- Apply use-cases and scenarios to drive the discovery of relationships (CRCs)
- Determine basic structural relationships that are tangible and conceptually obvious
- Identify relationships that can be transformed to generalization/specialization relationships. Use multiple inheritance only as needed
- Elaborate on basic relationships to specify intent – apply advanced features judiciously
- Use multiple diagrams to capture multiple abstractions (sets of relationships) to promote semantic clarity and specificity

**"iterative and evolutionary"** approach is key to the development of useful relationships (<u>why?</u>)

## References

- *Object-Oriented Analysis and Design with Applications, 3rd Edition, G. Booch, et. al. Addison Wesley, 2007*
- *OMG Unified Modeling Language Specification, http://www.omg.org/spec/, 2016*
- *The Unified Modeling Language User Guide, G. Booch, J. Rumbaugh, I. Jacobson, Addison Wesley Object Technology Series, 1999*
- *OMG Document Number formal/2015-03-01*

# UML: Behind the Scenes

Meta-models for structural specification languages

# Element Abstract Syntax

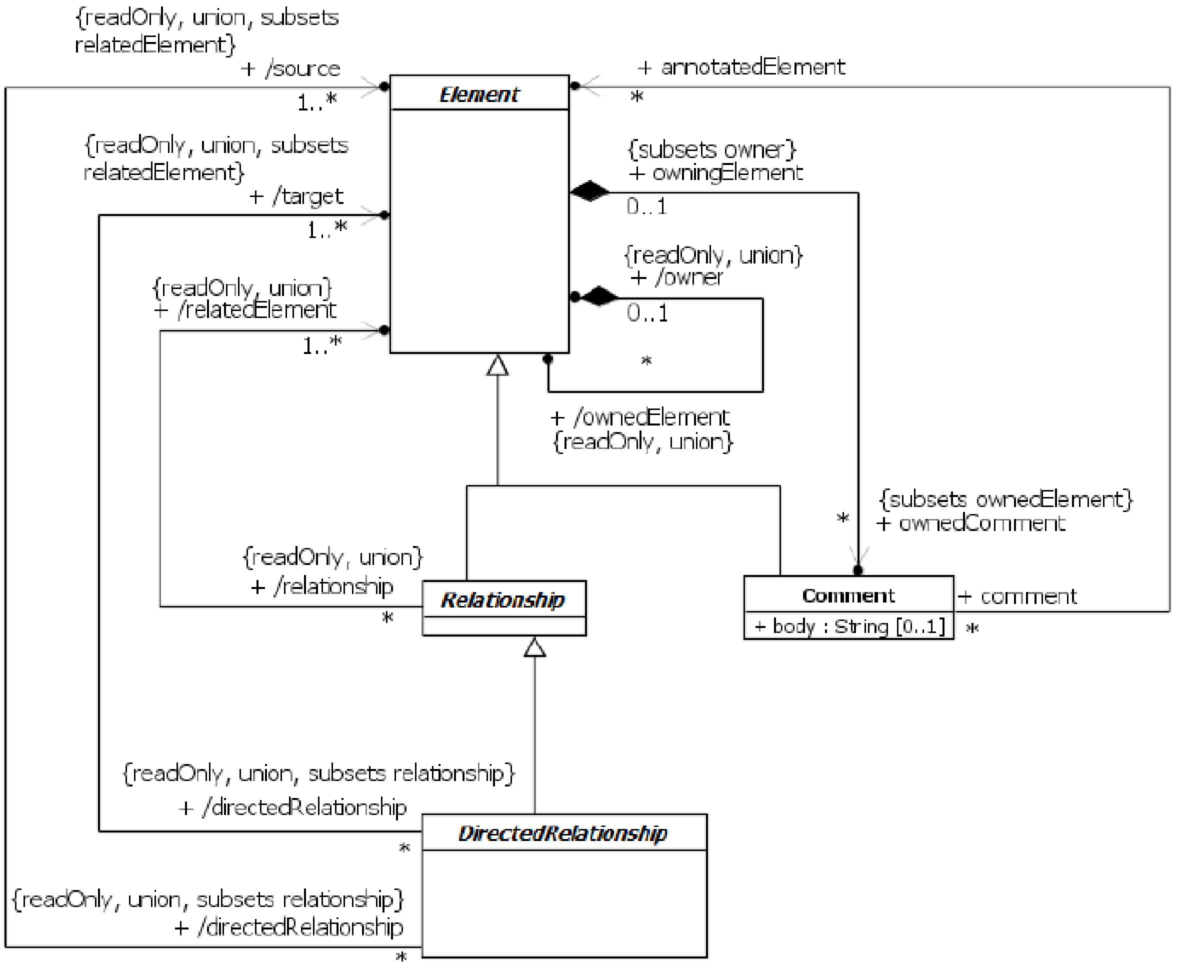
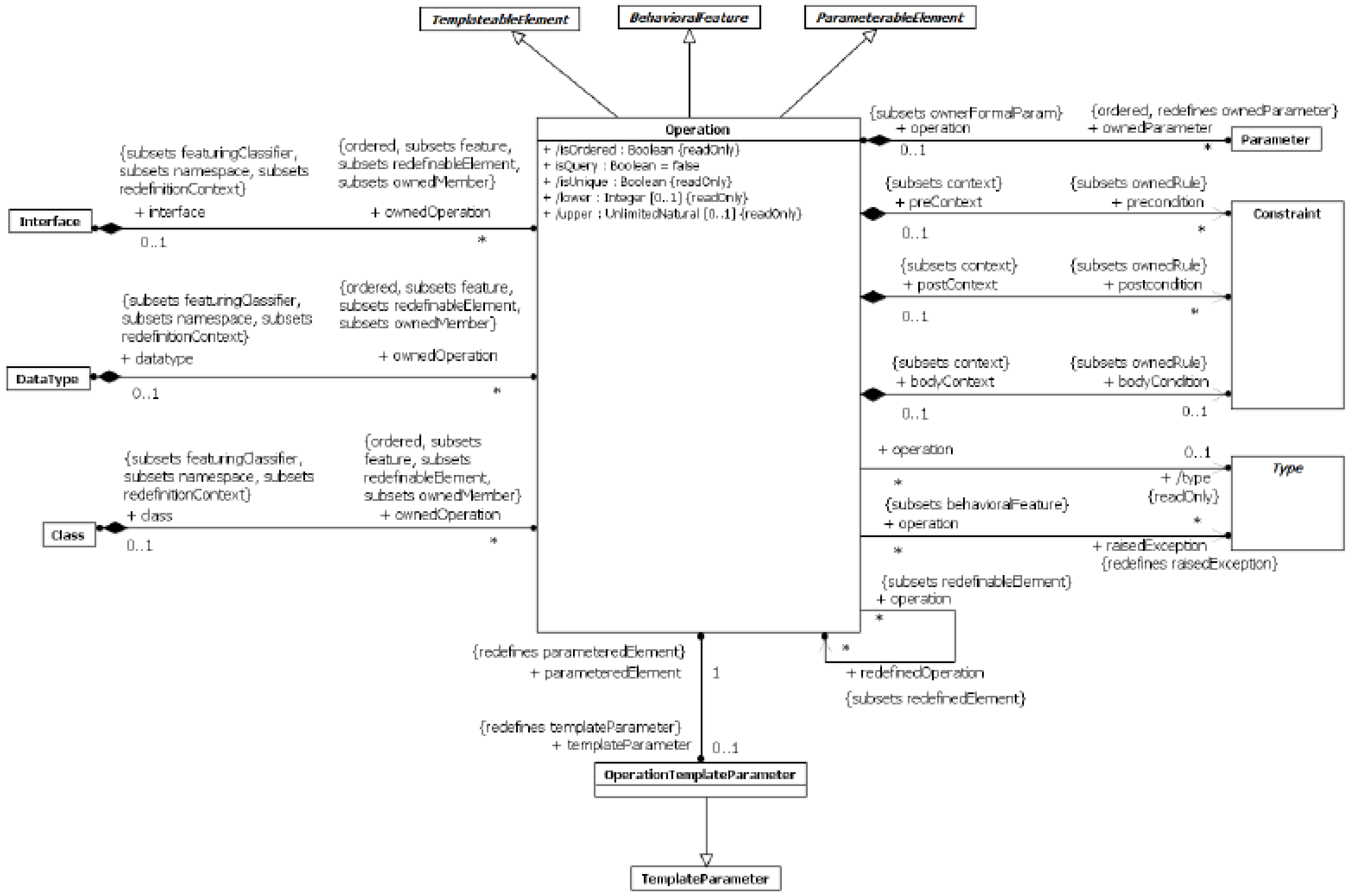


**Figure 7.1 Root**

# Operations Abstract Syntax



TemplateableElement
BehavioralFeature
ParameterableElement
Operation
+ /isOrdered : Boolean {readOnly}
+ isQuery : Boolean = false
+ /isUnique : Boolean {readOnly}
+ /lower : Integer [0..1] {readOnly}
+ /upper : UnlimitedNatural [0..1] {readOnly}
Interface
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+ interface
0..1
{ordered, subsets feature, subsets redefinableElement, subsets ownedMember}
+ ownedOperation
*
DataType
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+ datatype
0..1
{ordered, subsets feature, subsets redefinableElement, subsets ownedMember}
+ ownedOperation
*
Class
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+ class
0..1
{ordered, subsets feature, subsets redefinableElement, subsets ownedMember}
+ ownedOperation
*
{subsets ownerFormalParam}
+ operation
0..1
{ordered, redefines ownedParameter}
+ ownedParameter
*
Parameter
{subsets context}
+ preContext
0..1
{subsets ownedRule}
+ precondition
*
Constraint
{subsets context}
+ postContext
0..1
{subsets ownedRule}
+ postcondition
*
{subsets context}
+ bodyContext
0..1
{subsets ownedRule}
+ bodyCondition
0..1
+ operation
*
0..1
+ /type
{readOnly}
Type
{subsets behavioralFeature}
+ operation
*
*
+ raisedException
{redefines raisedException}
{subsets redefinableElement}
+ operation
*
*
+ redefinedOperation
{subsets redefinedElement}
{redefines parameteredElement}
+ parameteredElement
1
{redefines templateParameter}
+ templateParameter
0..1
OperationTemplateParameter
TemplateParameter

**Figure 9.13 Operations**

# Expression Abstract Syntax

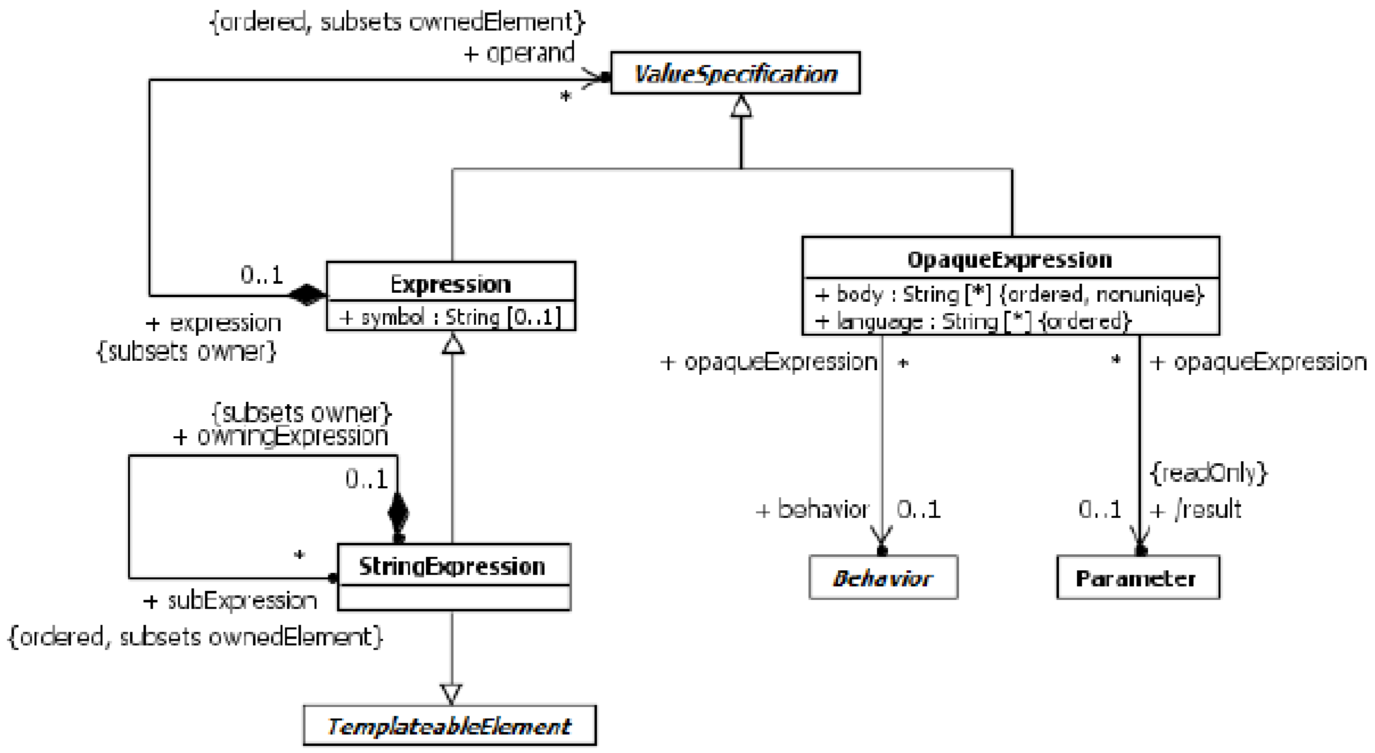


**Figure 8.2 Expressions**

# Interfaces Abstract Syntax

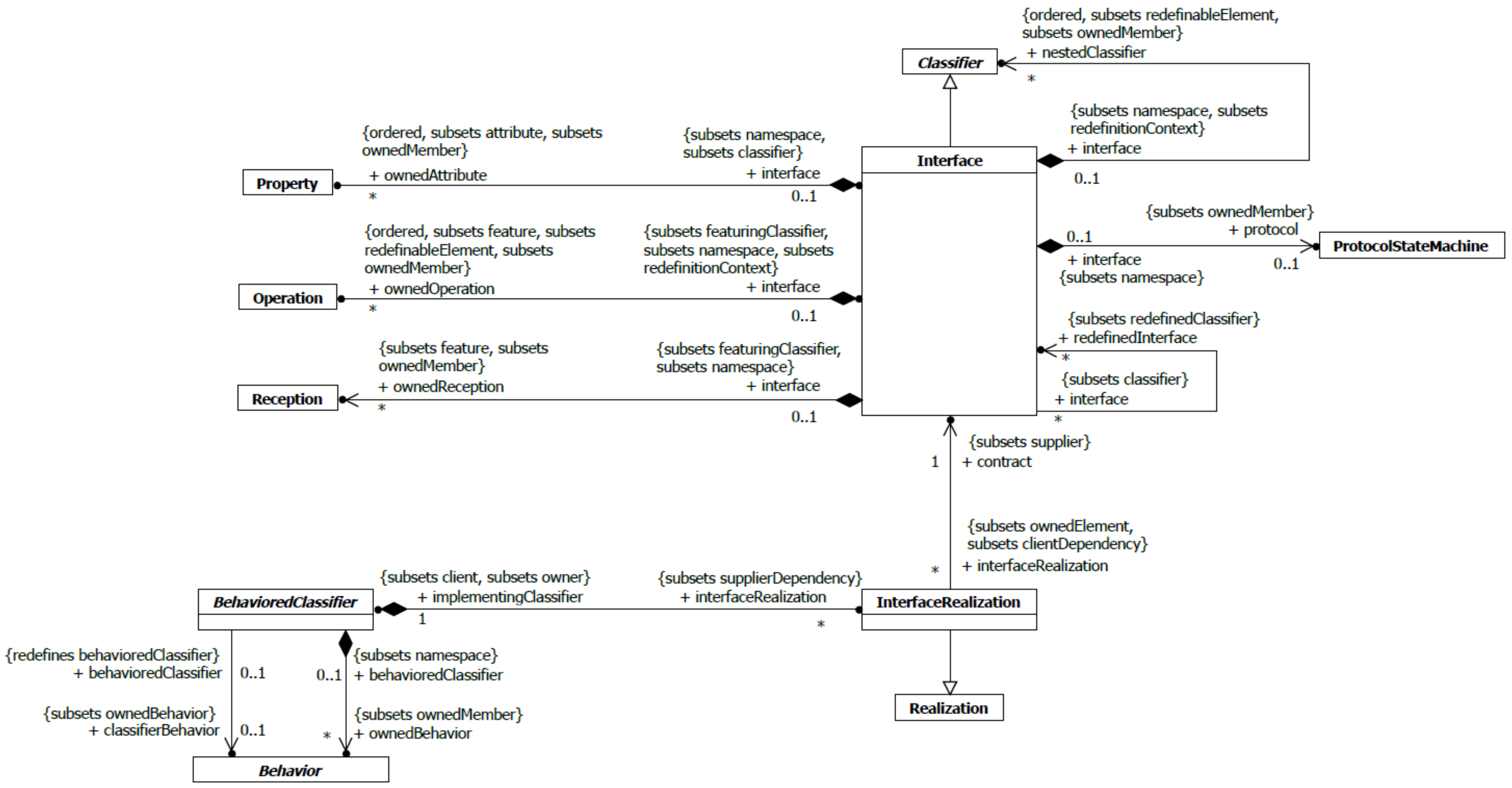


**Figure 10.7 Interfaces**

# Classifier Abstract Syntax

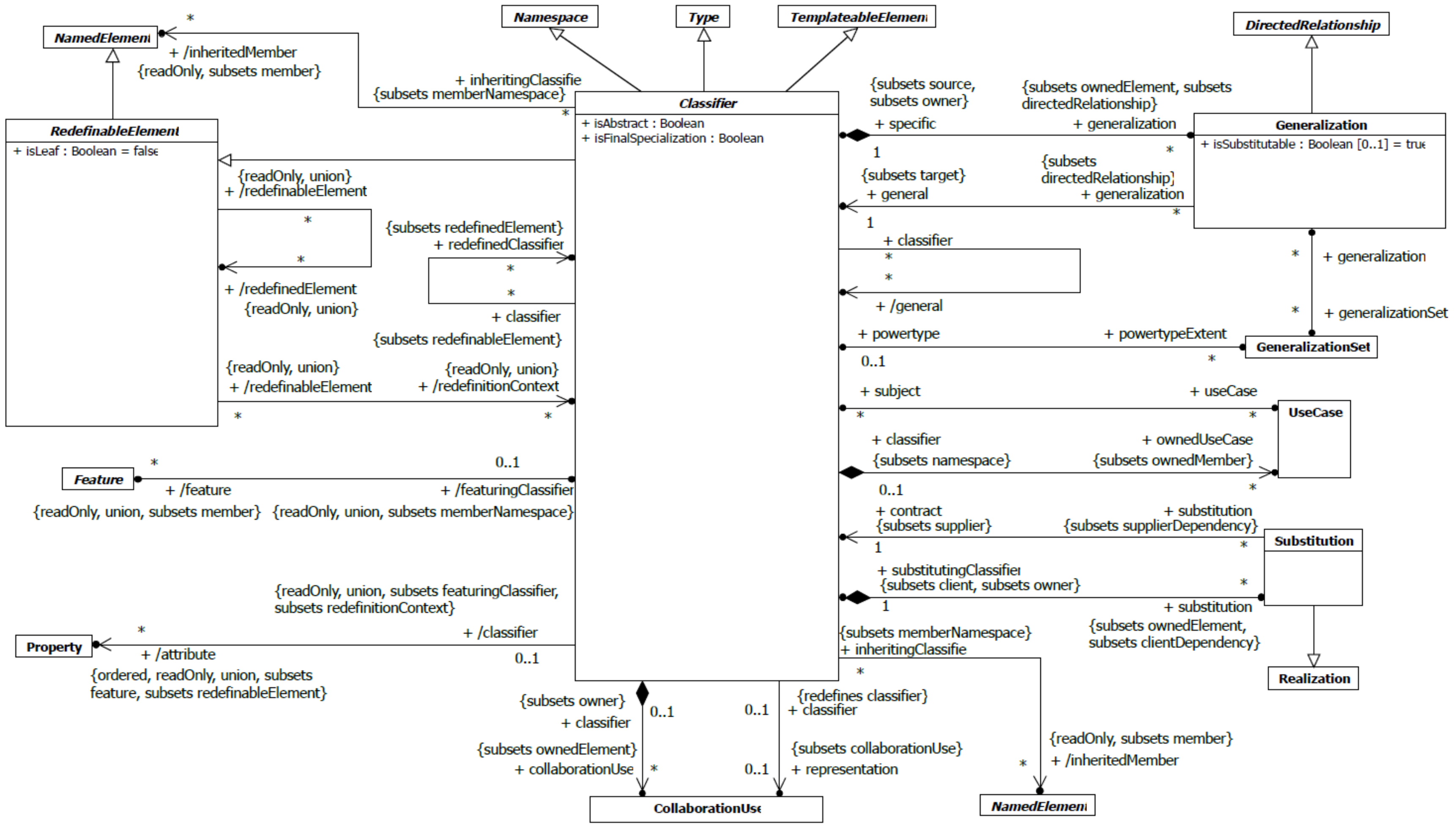


**Figure 9.1 Classifiers**

# Structured Abstract Syntax

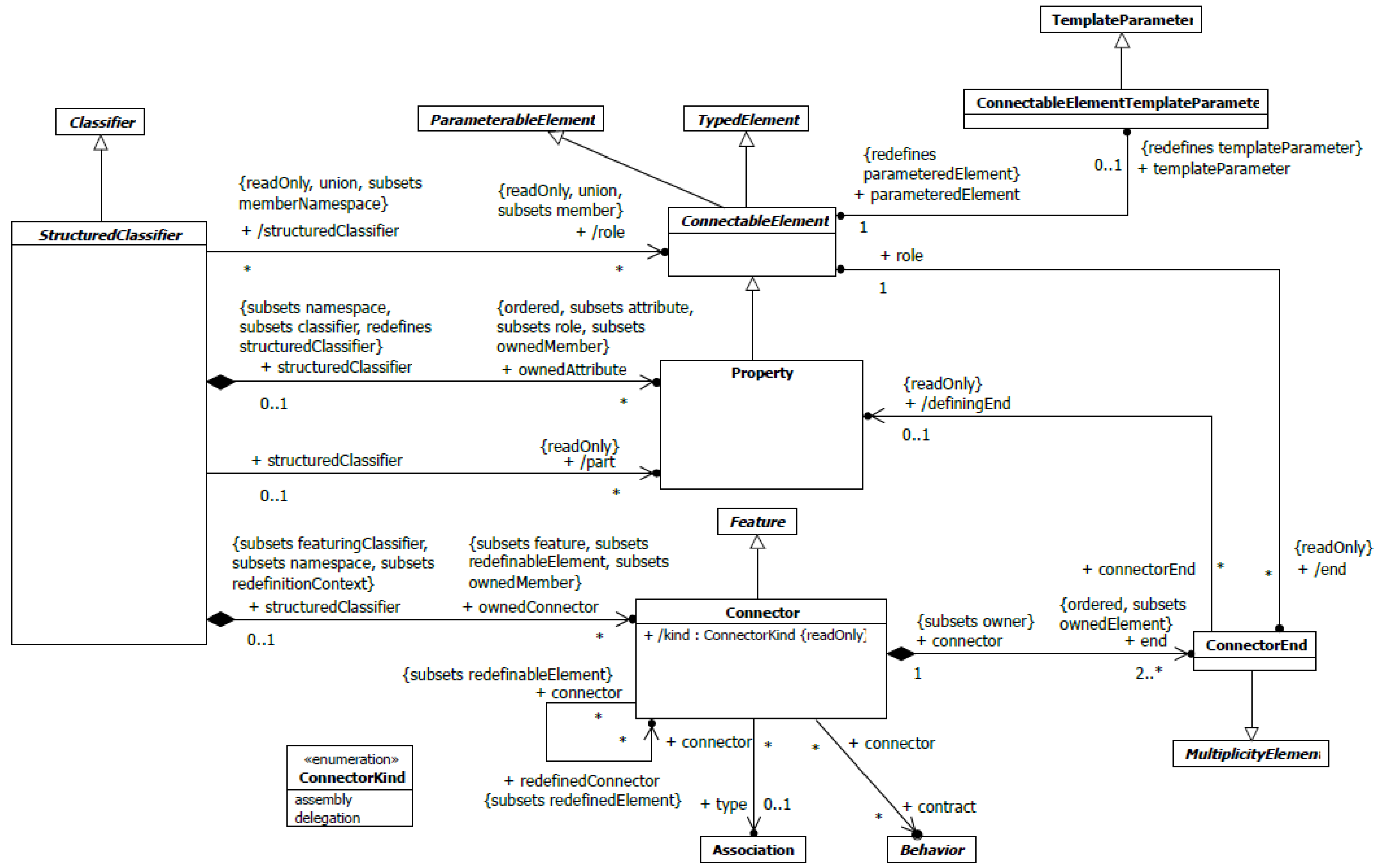


**Figure 11.1 Structured Classifiers**

## Generalization Sets Abstract Syntax

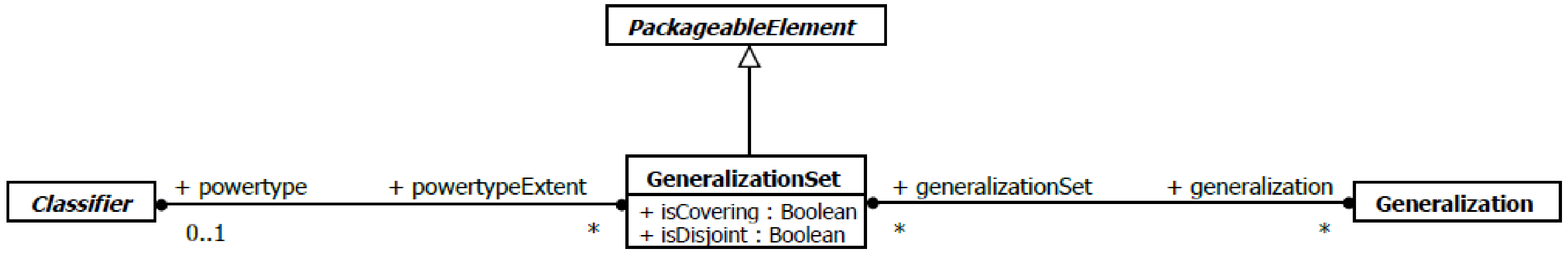


**Figure 9.14 Generalization Sets**

# Dependency Abstract Syntax

## 7.7.2 Abstract Syntax

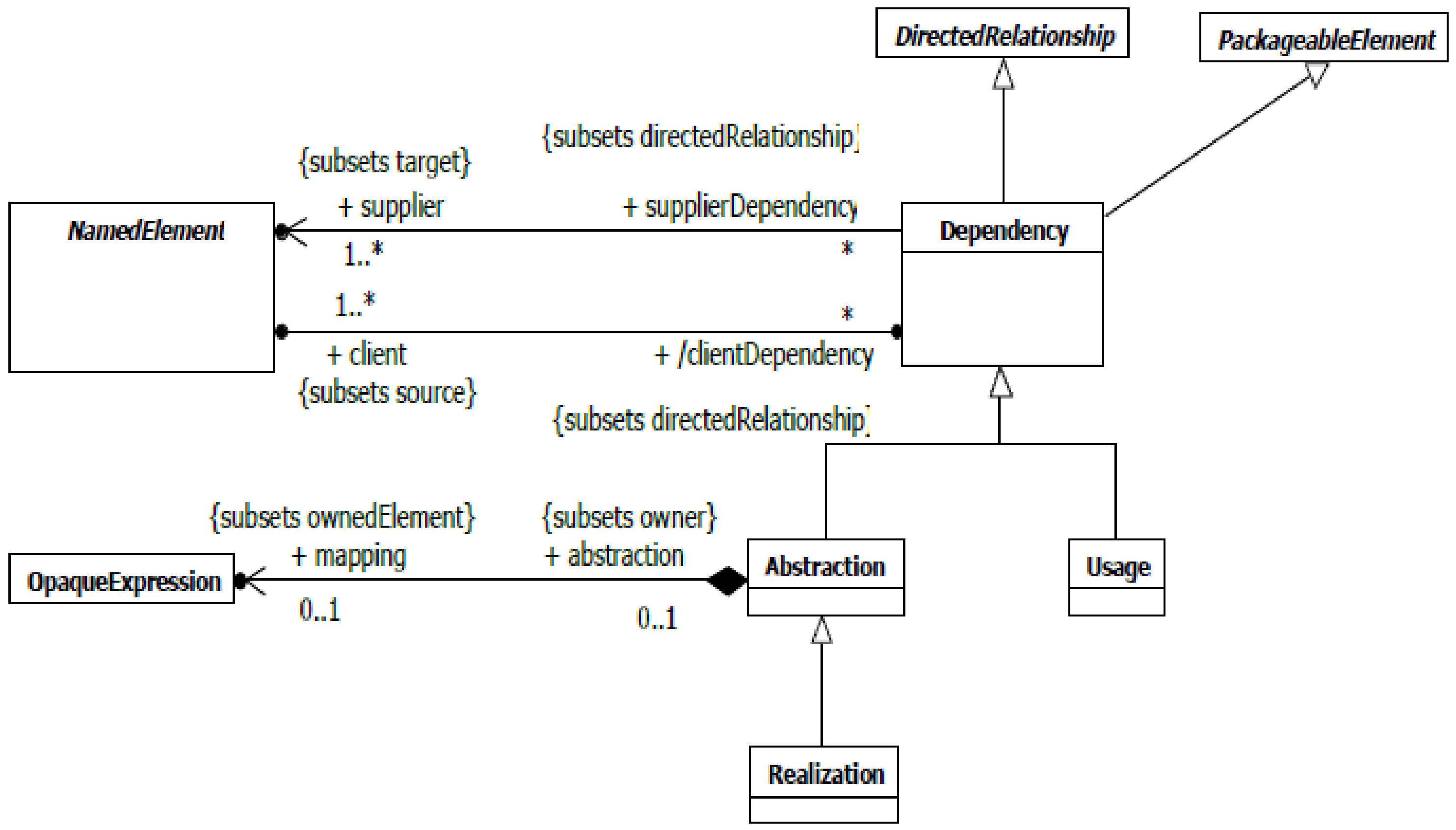


**Figure 7.17 Abstract syntax of dependencies**