Software Design, Modelling and Analysis in UML

Lecture 19: Inheritance II, Meta-Modelling

2012-02-08

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- Live Sequence Charts Semantics

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What's the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What's the effect of inheritance on LSCs, State Machines, System States?
  - What's the idea of Meta-Modelling?
- **Content:**
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics

Inheritance: Desired Semantics

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].
(**Liskov Substitution Principle** (LSP).)

"If for each object o_1 of type S there is an object o_2 of type T such that
for all programs P defined in terms of T,
the behavior of P is unchanged when o_1 is substituted for o_2
then S is a **subtype** of T."

S sub-type of T $:\iff \forall o_1 \in S\ \exists o_2 \in T\ \forall P_T \bullet [\![P_T]\!](o_1) = [\![P_T]\!](o_2)$

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which in the OO domain is closely related to inheritance:

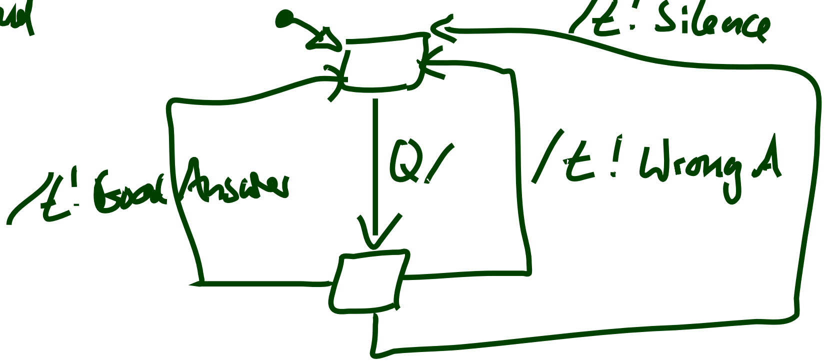
The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].
(**Liskov Substitution Principle** (LSP).)

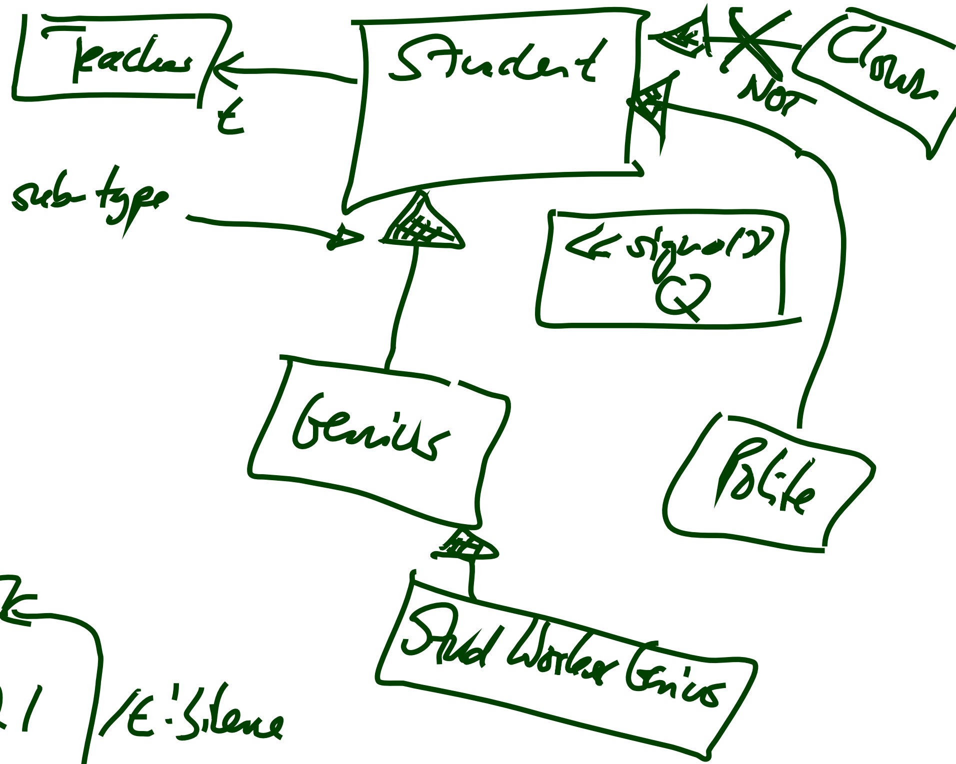
> "If for each object o_1 of type S there is an object o_2 of type T such that
> for all programs P defined in terms of T,
> **the behavior of P is unchanged** when o_1 is substituted for o_2
> then S is a **subtype** of T."

In other words: [Fischer and Wehrheim, 2000]

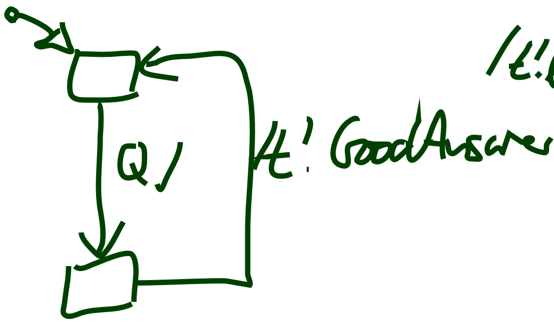
> "An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected,
> **without a client being able to tell the difference**."

SM_{Stud}



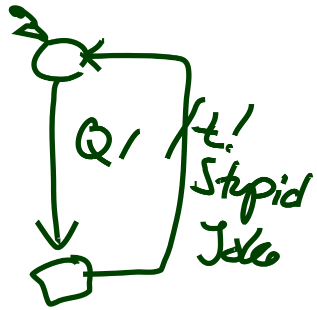


SM_{Genius}

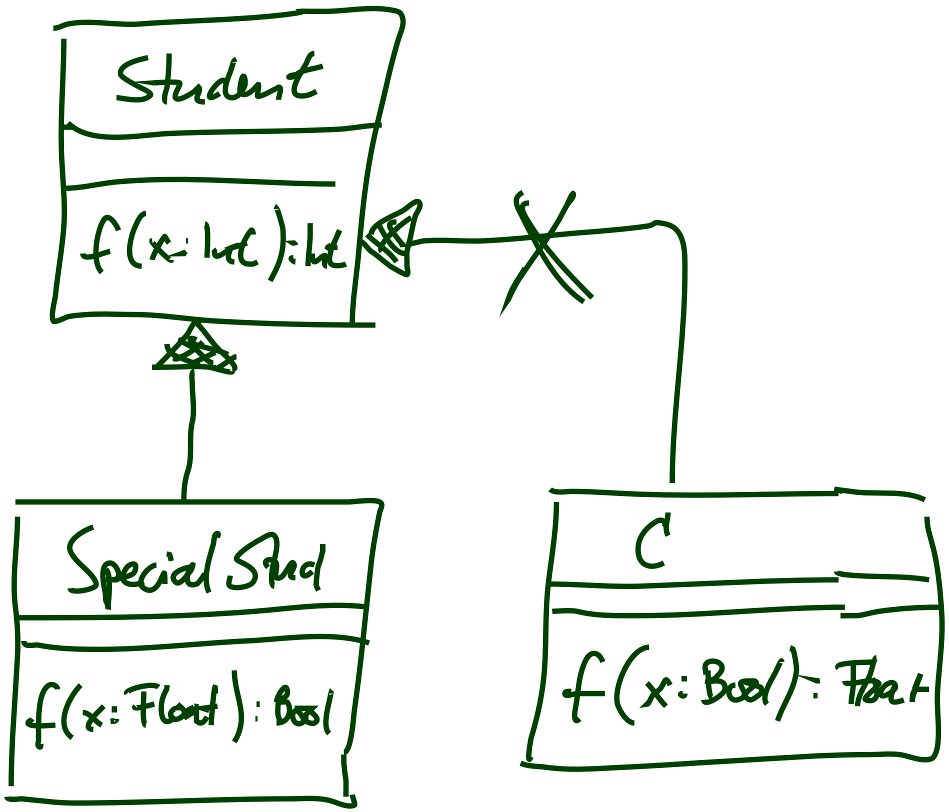


SM_{Polite}

SM_{Clown}



$SM_{StudWorkerGenius}$


Student
f(x: Int): Int
SpecialStud
f(x: Float): Bool
C
f(x: Bool): Float

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994]. (**Liskov Substitution Principle** (LSP).)

> "If for each object o_1 of type S there is an object o_2 of type T such that
> for all programs P defined in terms of T,
> **the behavior of P is unchanged** when o_1 is substituted for o_2
> then S is a **subtype** of T."

In other words: [Fischer and Wehrheim, 2000]

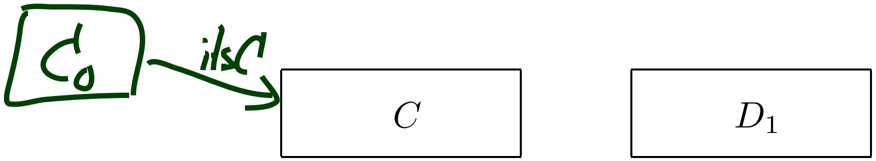
> "An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected,
> **without a client being able to tell the difference**."

So, what's "**usable**"? Who's a "**client**"? And what's a "**difference**"?

What Does [Fischer and Wehrheim, 2000] Mean for UML?

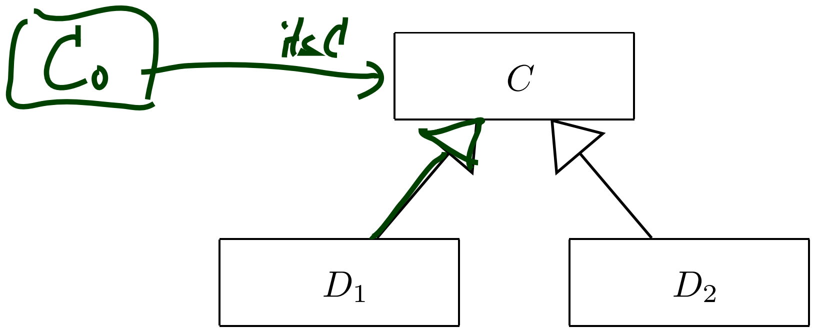
> "An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected, without **a client** being able to tell the **difference**."

- Wanted: sub-typing for UML.
- With



  we don't even have usability.

- It would be nice, if the well-formedness rules and semantics of

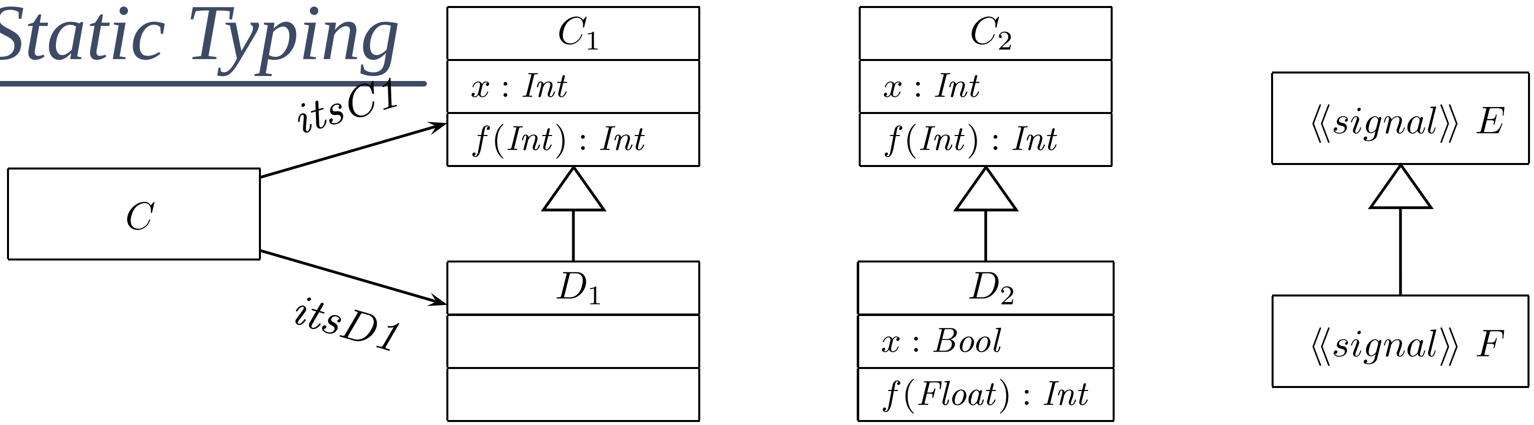


  would ensure D_1 **is a sub-type** of C:
  - that D_1 objects can be used interchangeably by everyone who is using C's,
  - is not able to tell the difference (i.e. see unexpected behaviour).

“...shall be usable...” for UML

Easy: Static Typing

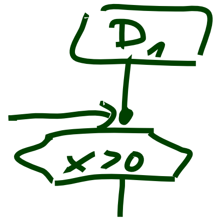
Given:



Wanted:

- $x > 0$ also **well-typed** for D_1

e.g.



e.g. context D_1 inv : $x > 0$

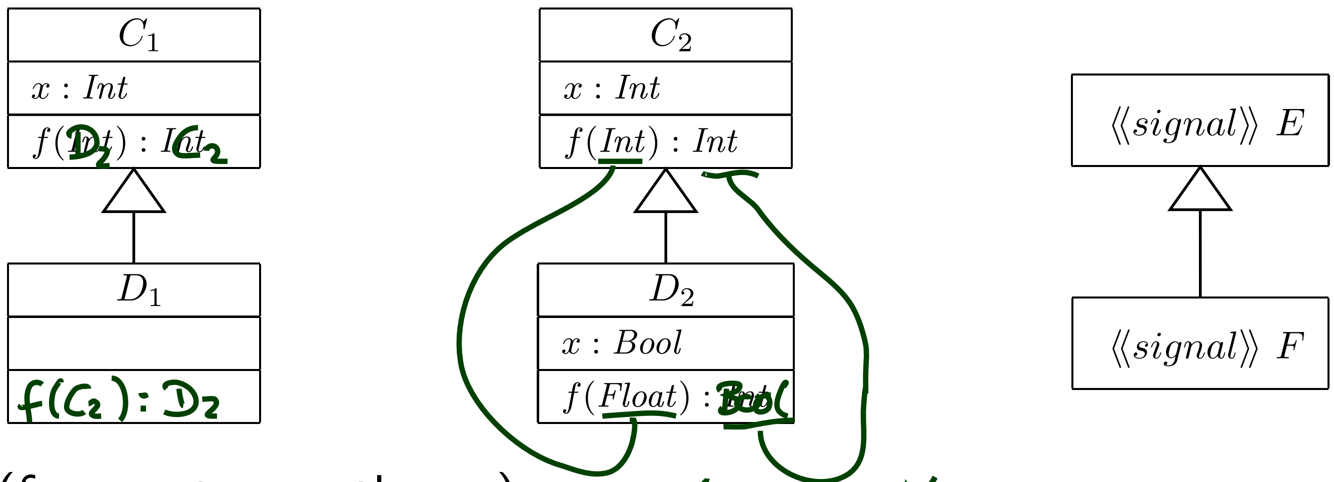
- assignment $itsC1 := itsD1$ being **well-typed** (not other way round)
- $itsD1.x = 0$, $itsD1.f(0)$, $itsD1 \,!\, F$
  being well-typed (and doing the right thing).

Approach:

- Simply define it as being well-typed,
  adjust system state definition to do the right thing.

e.g. $v := expr$ is well typed if $v : \tau_{C'}$, $expr : \tau_{D_1}$
and $C' \lhd^* D_1$

Static Typing Cont'd



Notions (from category theory):

accepts more general

provides sth. more specialised

- **invariance**,
- **covariance**,
- **contravariance**.

We could call, e.g. a method, **sub-type preserving**, if and only if it

- accepts **more general** types as input (**contravariant**),
- provides a **more specialised** type as output (**covariant**).

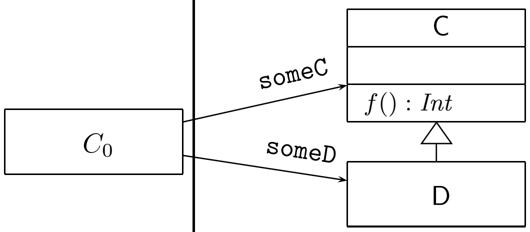
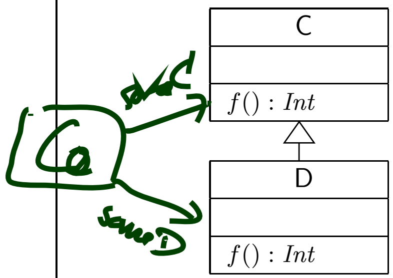
This is a notion used by many programming languages — and easily type-checked.

Excursus: Late Binding of Behavioural Features

Late Binding

What transformer applies in what situation? (Early (compile time) binding.)

type of link determines which implementation is used (not caring for what the object really "is")

	f not overridden in D	f overridden in D	value of someC/ someD
			
someC -> f()	C::f()	C::f()	u_1 : C
someD -> f()	C::f()	D::f()	u_2 : D
someC -> f()	C::f()	C::f()	u_2 : D

What one could want is something different: (Late binding.)

type of object determines which impl. is used

someC -> f()	C::f()	C::f()	u_1 : C
someD -> f()	C::f()	D::f()	u_2 : D
someC -> f()	C::f()	D::f()	u_2 : D

– 19 – 2012-02-08 – Slatebind –

Late Binding in the Standard and Programming Lang.

- In **the standard**, Section 11.3.10, "CallOperationAction":

  "**Semantic Variation Points**
  The mechanism for determining the method to be invoked as a result of a call operation is unspecified." [OMG, 2007b, 247]

- In **C++**,
  - methods are by default "(early) compile time binding",
  - can be declared to be "late binding" by keyword "`virtual`",
  - the declaration applies to all inheriting classes.

- In **Java**,
  - methods are "late binding";
  - there are patterns to imitate the effect of "early binding"

Exercise: What could have driven the designers of C++ to take that approach?

Note: late binding typically applies only to **methods**, **not** to **attributes**.
(But: getter/setter methods have been invented recently.)

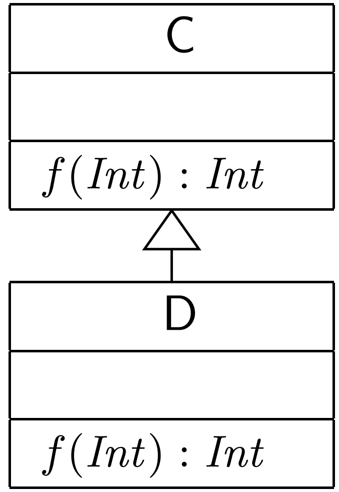
Back to the Main Track: "...tell the difference..." for UML

With Only Early Binding...

- ...we're **done** (if we realise it correctly in the framework).
- Then
  - if we're calling method f of an object u,
  - which is an instance of D with $C \trianglelefteq^* D$
  - via a C-link, $C::f()$ will be called
  - then we (by definition) only see and change the C-part.
  - We cannot tell whether u is a C or an D instance.

So we immediately also have behavioural/dynamic subtyping.

Difficult: Dynamic Subtyping



- $C{::}f$ and $D{::}f$ are **type compatible**, but D is **not necessarily** a **sub-type** of C.

- **Examples**: (C++)

```
int C::f(int) {
   return 0;
};
```

vs.

```
int D::f(int) {
   return 1;
};
```

Sub-Typing Principles Cont'd

- In the standard, Section 7.3.36, "**Operation**":

  > "**Semantic Variation Points**
  > [...] When operations are redefined in a specialization, rules regarding **invariance**, **covariance**, or **contravariance** of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations." [OMG, 2007a, 106]

- So, better: call a method **sub-type preserving**, if and only if it
  (i) accepts **more input values** (**contravariant**),
  (ii) on the **old values**, has **fewer behaviour** (**covariant**).

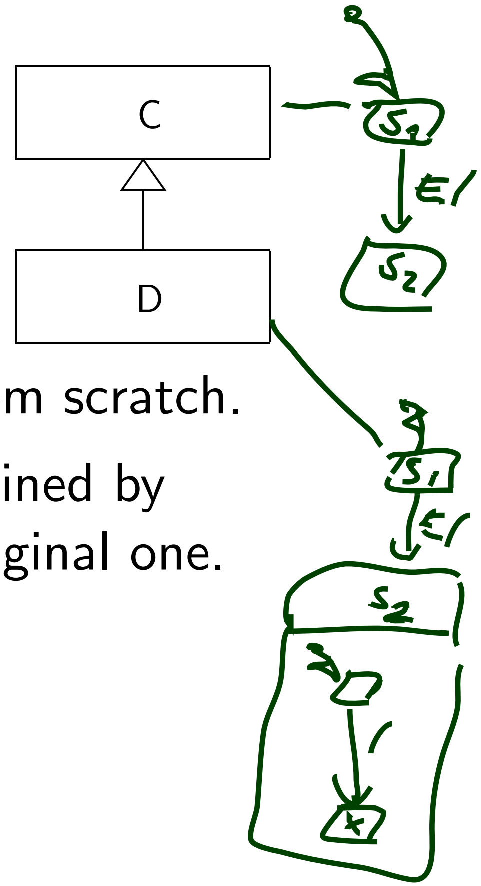
  Note: ~~This~~ (ii) is no longer a matter of simple type-checking!

- And not necessarily the end of the story:
  - One could, e.g. want to consider execution time.
  - Or, like [Fischer and Wehrheim, 2000], relax to "fewer observable behaviour", thus admitting the sub-type to do more work on inputs.

  Note: "testing" differences depends on the **granularity** of the semantics.

- **Related**: "has a weaker pre-condition," (**contravariant**),
  "has a stronger post-condition." (**covariant**).

Ensuring Sub-Typing for State Machines



- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.

- But the state machine of a sub-class **cannot** be drawn from scratch.
- Instead, the state machine of a sub-class can only be obtained by applying actions from a **restricted** set to a copy of the original one.

  Roughly (cf. User Guide, p. 760, for details),
  - add things into (hierarchical) states,
  - add more states,
  - attach a transition to a different target (limited).

- They **ensure**, that the sub-class is a **behavioural sub-type** of the super class. (But method implementations can still destroy that property.)

- Technically, the idea is that (by late binding) only the state machine of the most specialised classes are running.

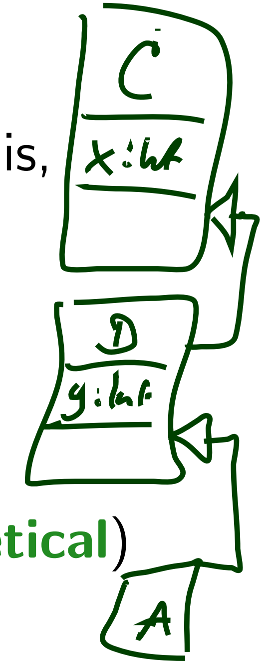
  By knowledge of the framework, the (code for) state machines of super-classes is still accessible — but using it is hardly a good idea...

Towards System States

Wanted: a formal representation of "if $C \preceq D$ then D '**is a**' C", that is,

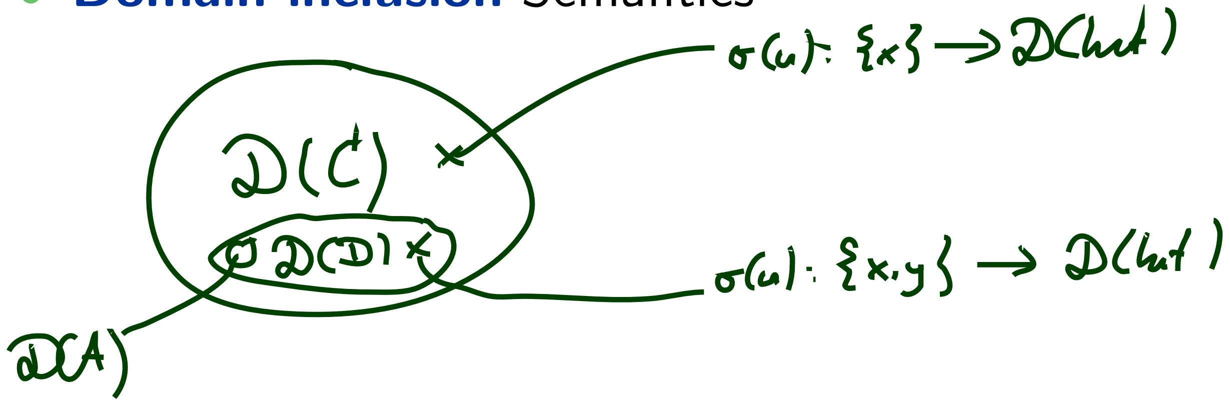
(i) D has the same attributes and behavioural features as C, and

(ii) D objects (identities) can replace C objects.

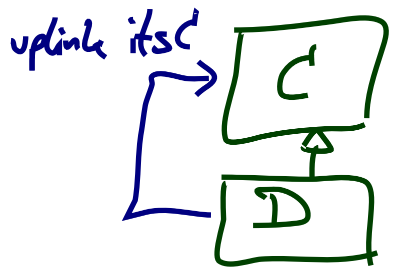


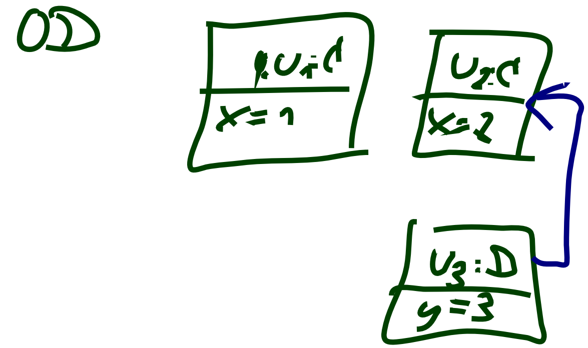
We'll discuss **two approaches** to semantics:

- **Domain-inclusion** Semantics (more **theoretical**)



- **Uplink** Semantics (more **technical**)





someD.x $\overset{\text{rewrite}}{\rightsquigarrow}$ someD.uplink itsC.x

Meta-Modelling: Idea and Example

Meta-Modelling: Why and What

- **Meta-Modelling** is one major prerequisite for understanding
  - the standard documents [OMG, 2007a, OMG, 2007b], and
  - the MDA ideas of the OMG.

- The idea is **simple**:
  - if a **modelling language** is about modelling **things**,
  - and if UML models are and comprise **things**,
  - then why not **model** those in a modelling language?

- In other words:

  Why not have a model $\mathcal{M}_U$ such that

  - the set of legal instances of $\mathcal{M}_U$

  is

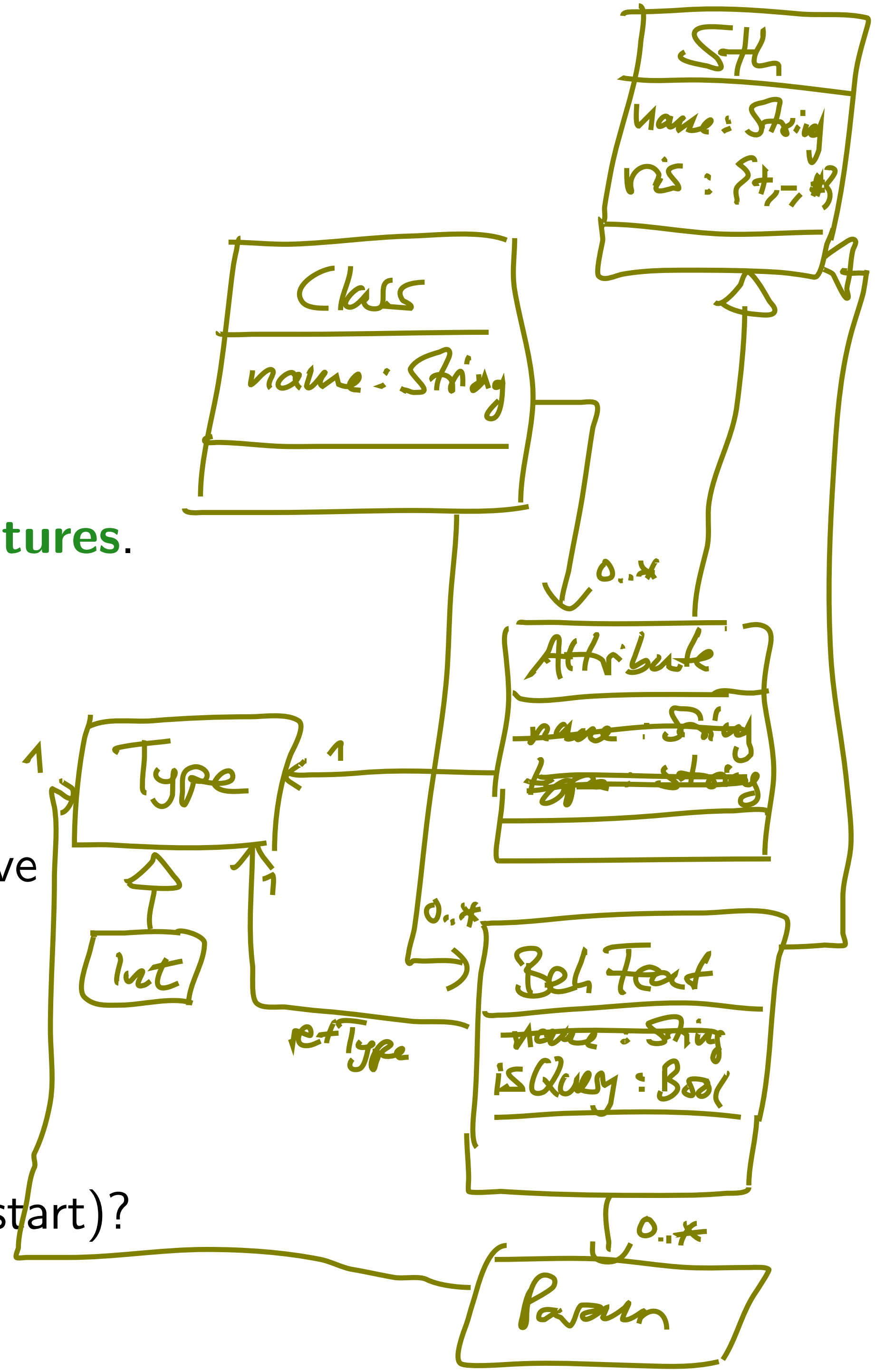
  - the set of well-formed (!) UML models.

Meta-Modelling: Example

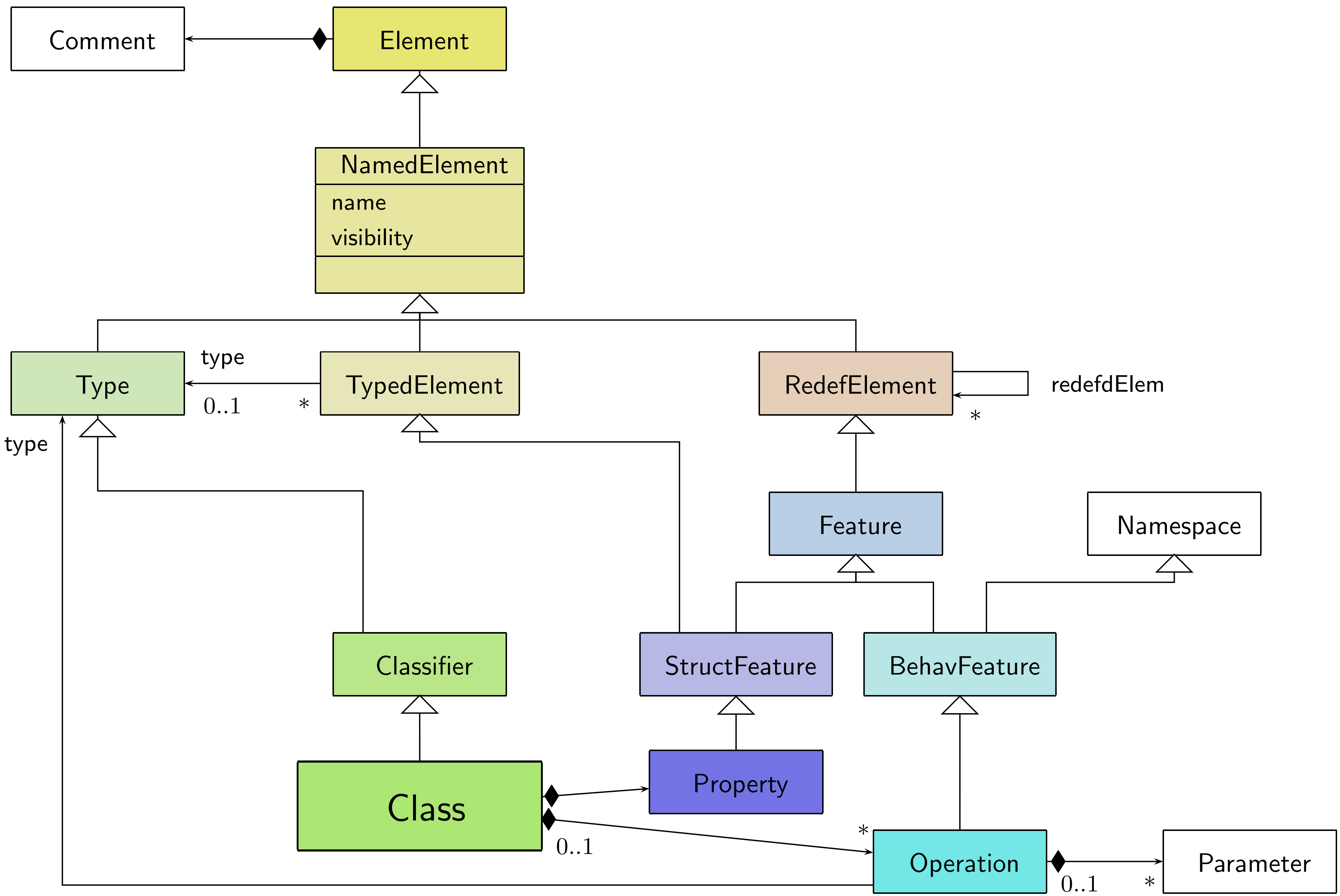
- For example, let's consider a class.
- A **class** has (on a superficial level)
  - a **name**,
  - any number of **attributes**,
  - any number of **behavioural features**.

  Each of the latter two has
  - a **name** and
  - a **visibility**.

  Behavioural features in addition have
  - a boolean attribute **isQuery**,
  - any number of parameters,
  - a return type.
- Can we model this (in UML, for a start)?



UML Meta-Model: Extract



Classes [OMG, 2007b, 32]

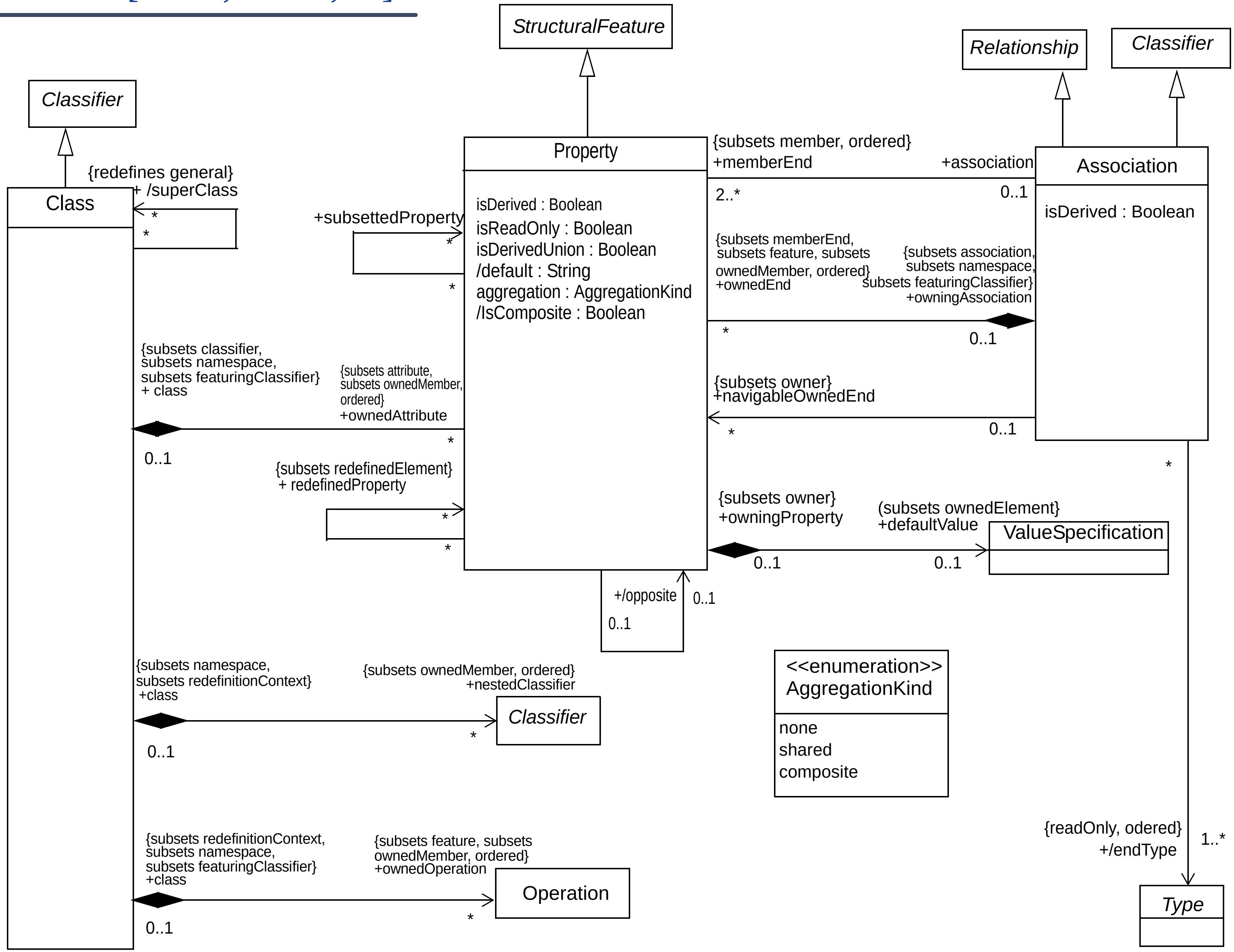


Figure 7.12 - Classes diagram of the Kernel package

Operations [OMG, 2007b, 31]

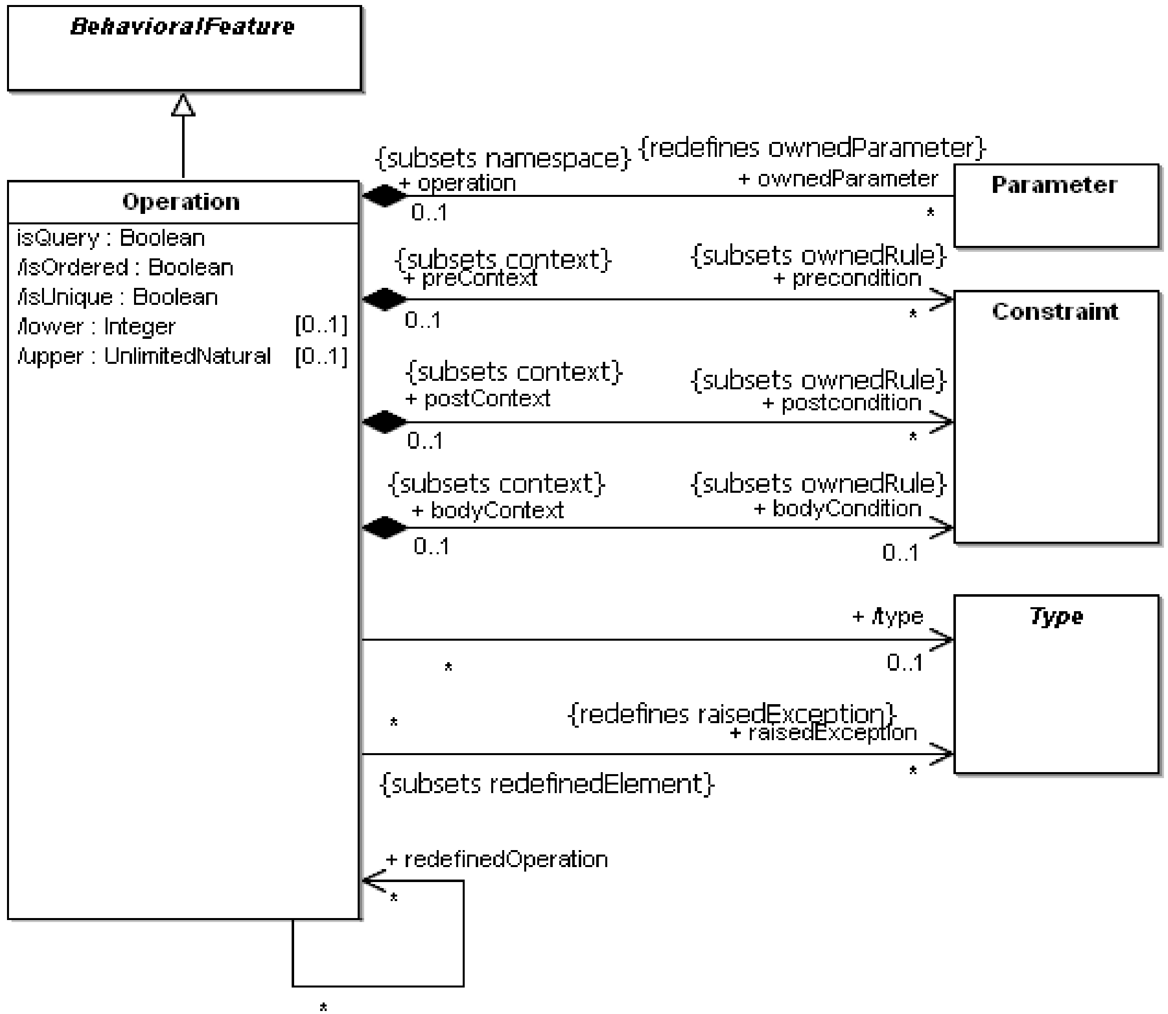


Figure 7.11 - Operations diagram of the Kernel package

Operations [OMG, 2007b, 30]

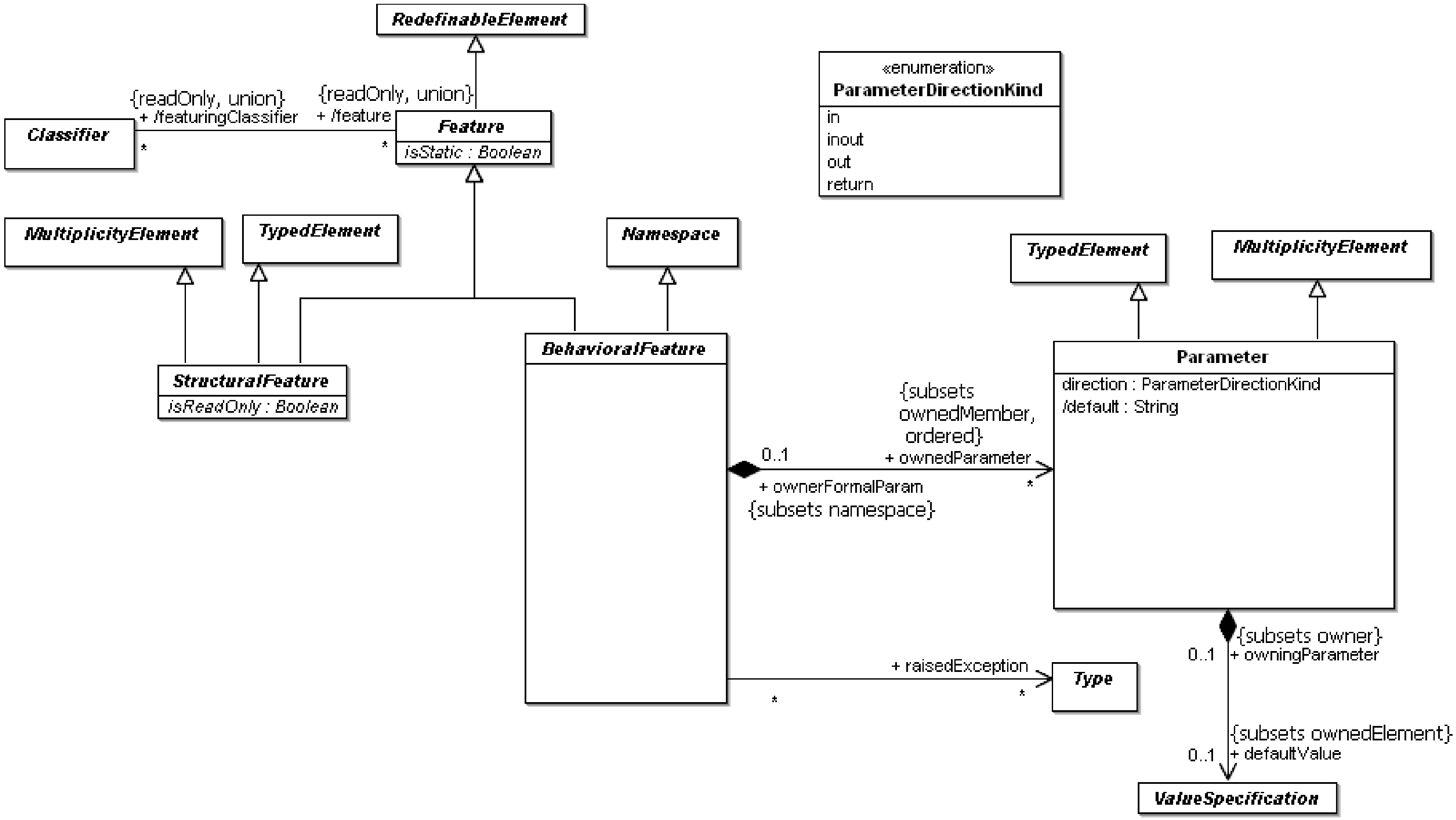


Figure 7.10 - Features diagram of the Kernel package

Classifiers [OMG, 2007b, 29]

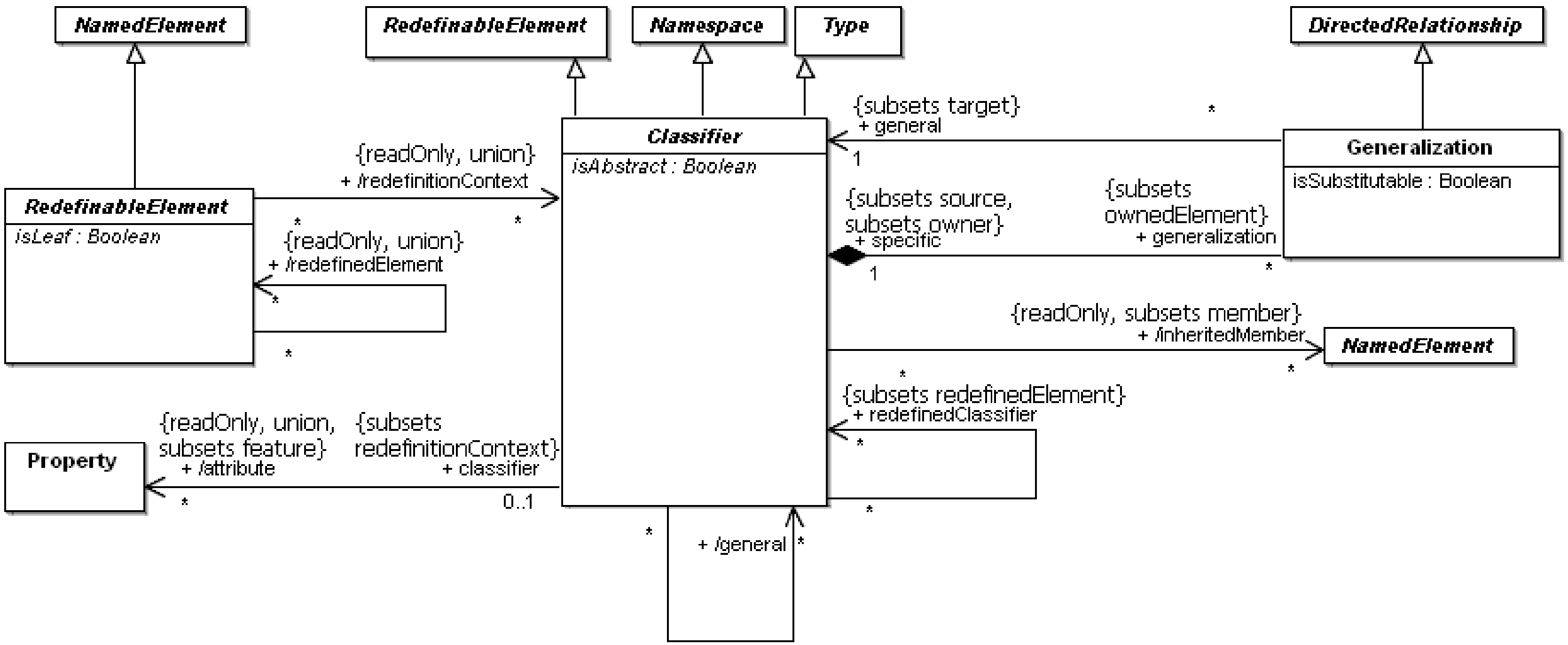


Figure 7.9 - Classifiers diagram of the Kernel package

Namespaces [OMG, 2007b, 26]

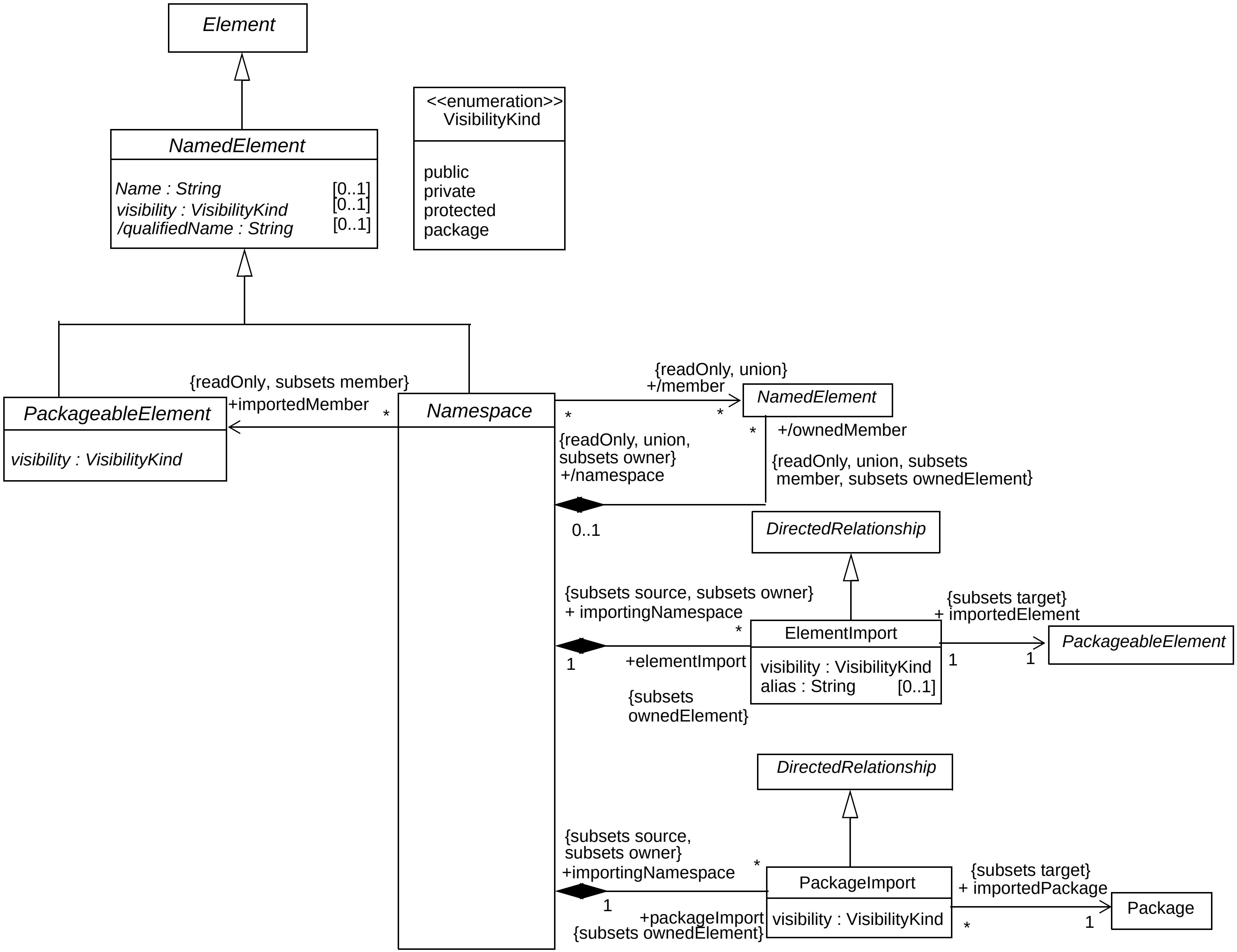


Figure 7.4 - Namespaces diagram of the Kernel package

Root Diagram [OMG, 2007b, 25]

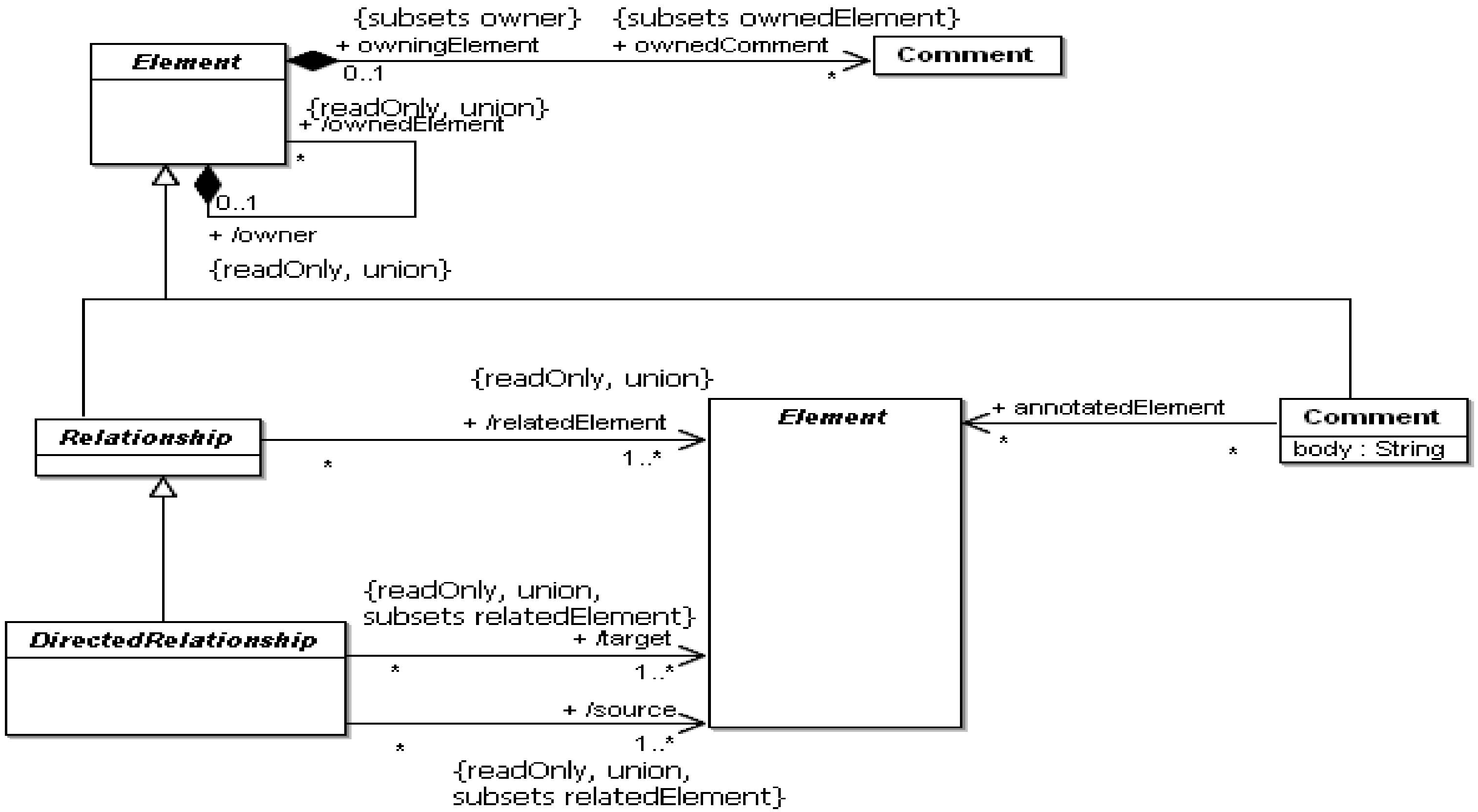


Figure 7.3 - Root diagram of the Kernel package

Interesting: Declaration/Definition [OMG, 2007b, 424]

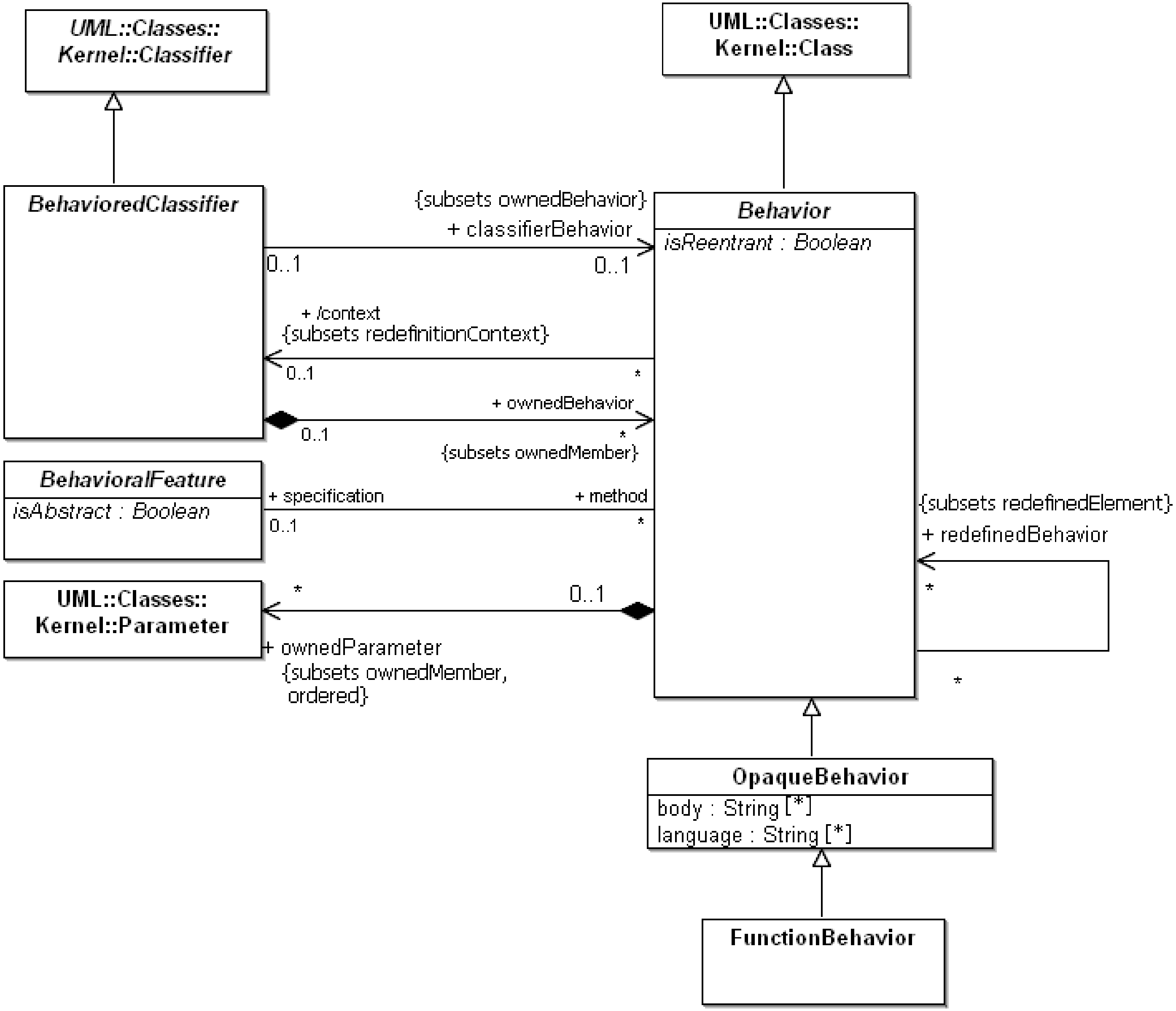


Figure 13.6 - Common Behavior

UML Architecture [?, 8]

- Meta-modelling has already been used for UML 1.x.

- For UML 2.0, the request for proposals (RFP) asked for a separation of concerns:

  Infrastructure and **Superstructure**.

- **One reason**:
  sharing with MOF (see later) and, e.g., CWM.

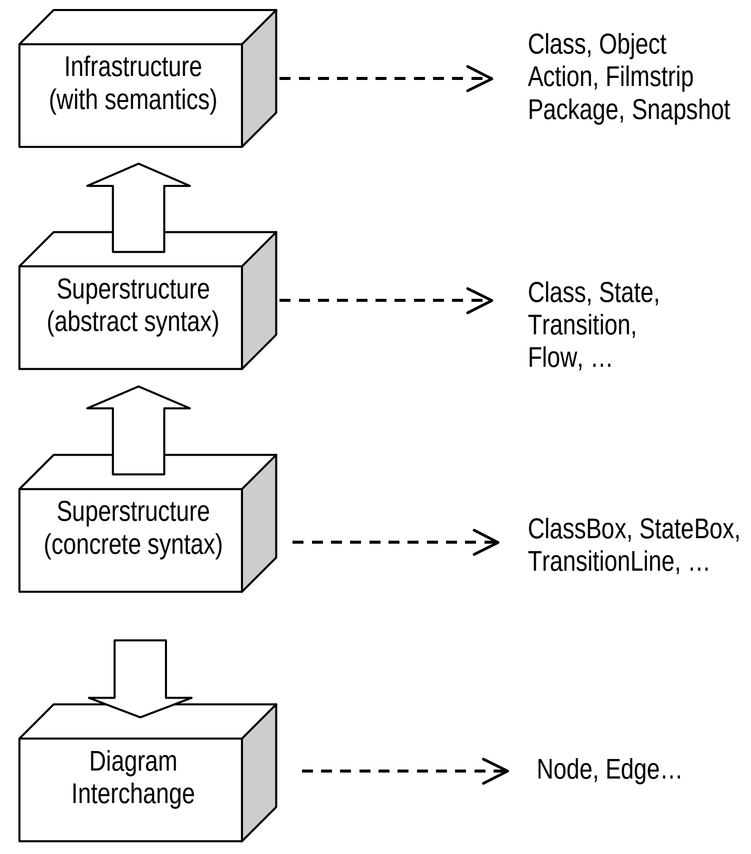
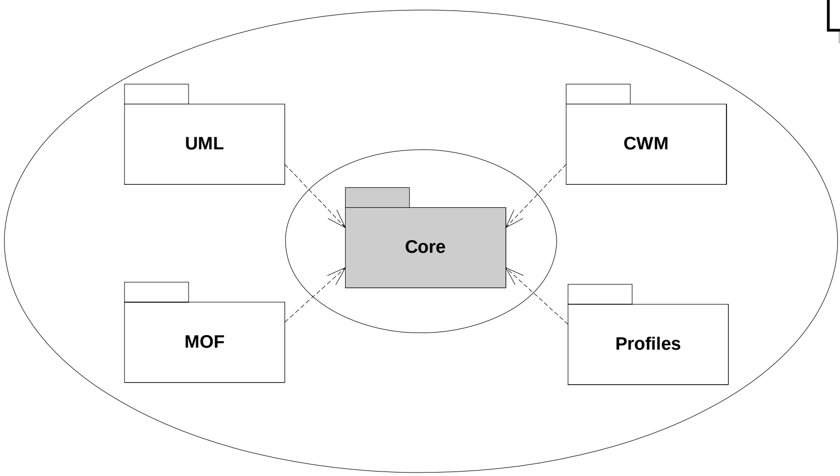


Figure 0-1 *Overview of architecture*



UML Superstructure Packages [OMG, 2007a, 15]

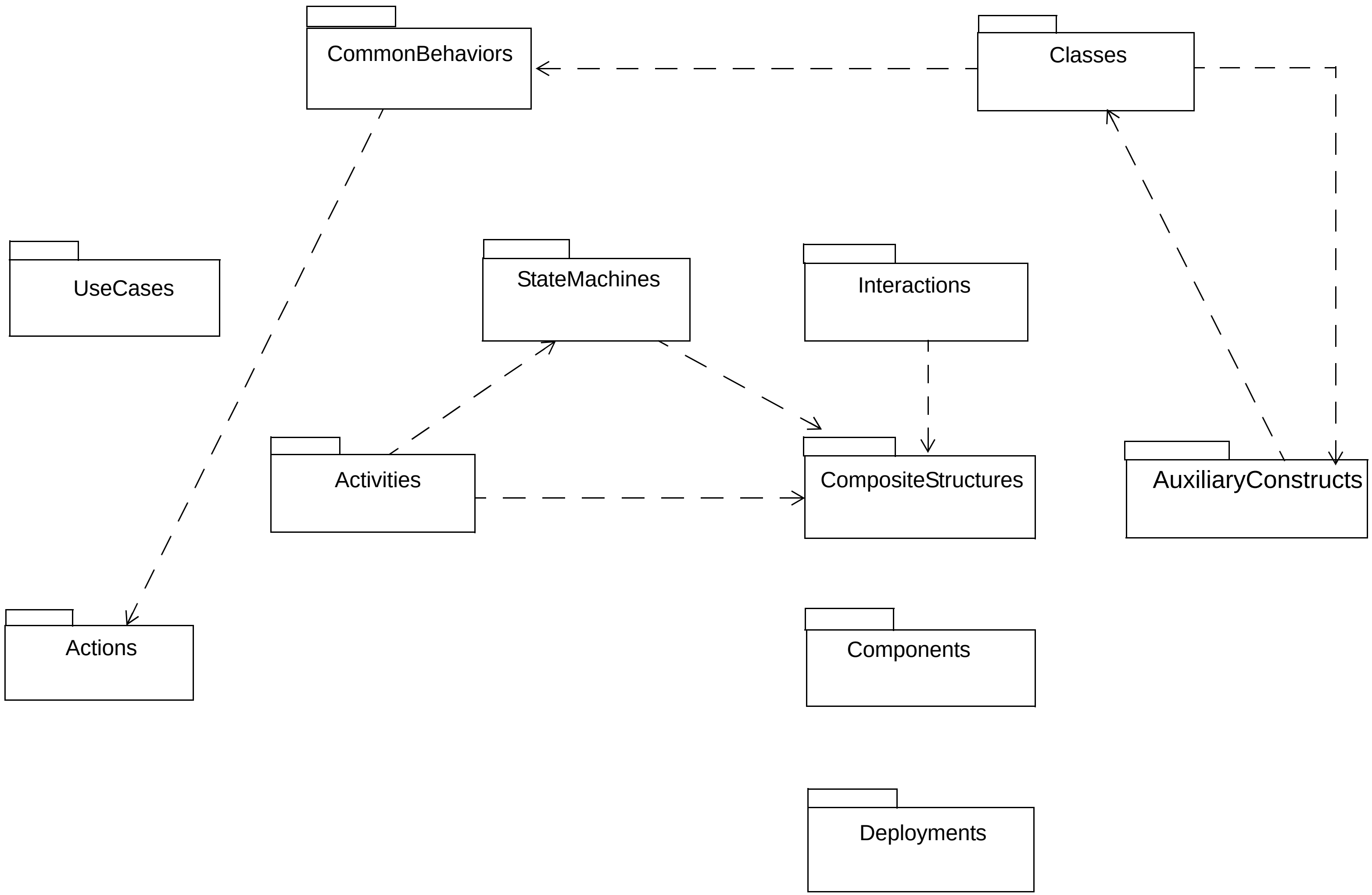
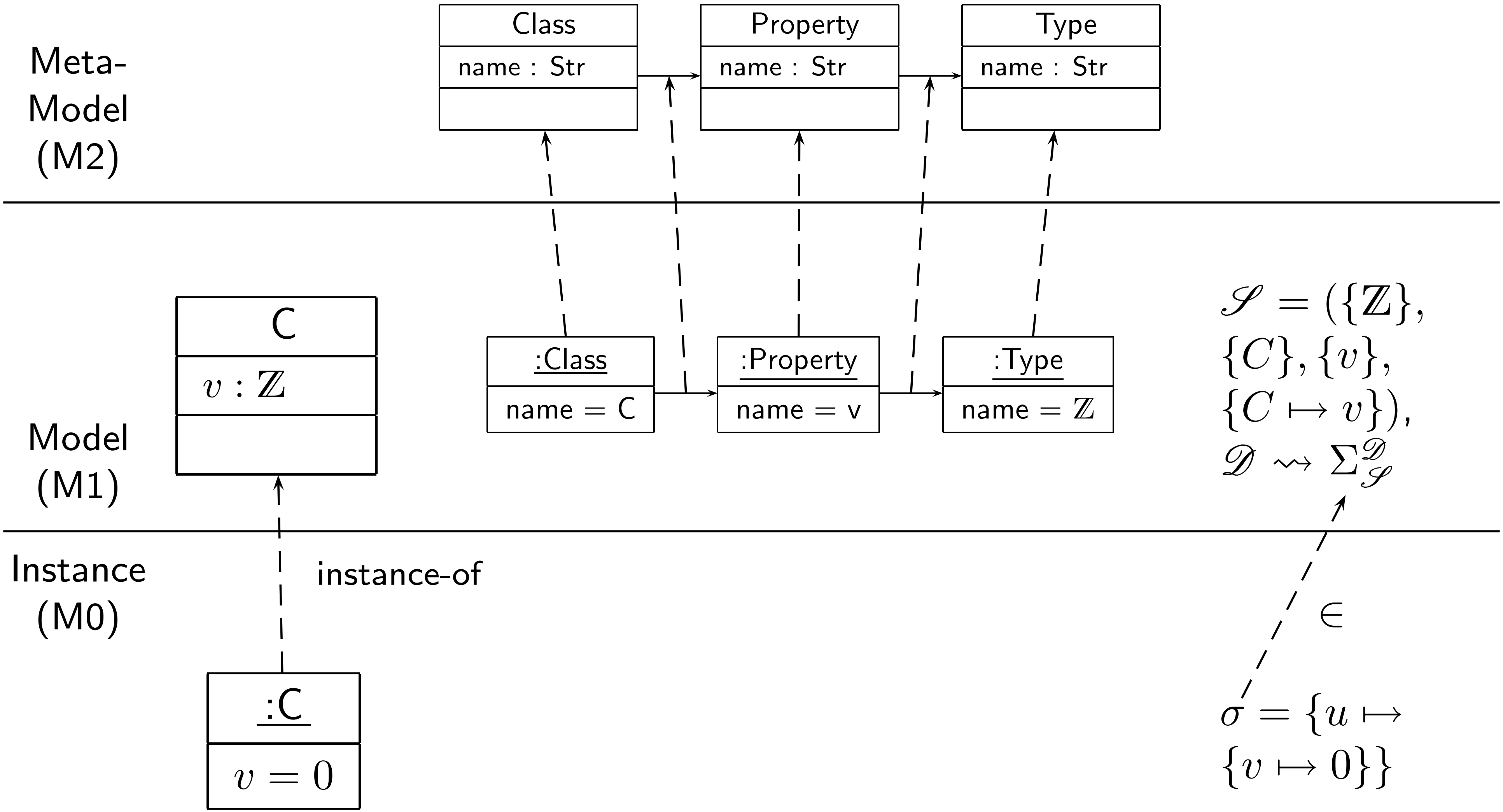


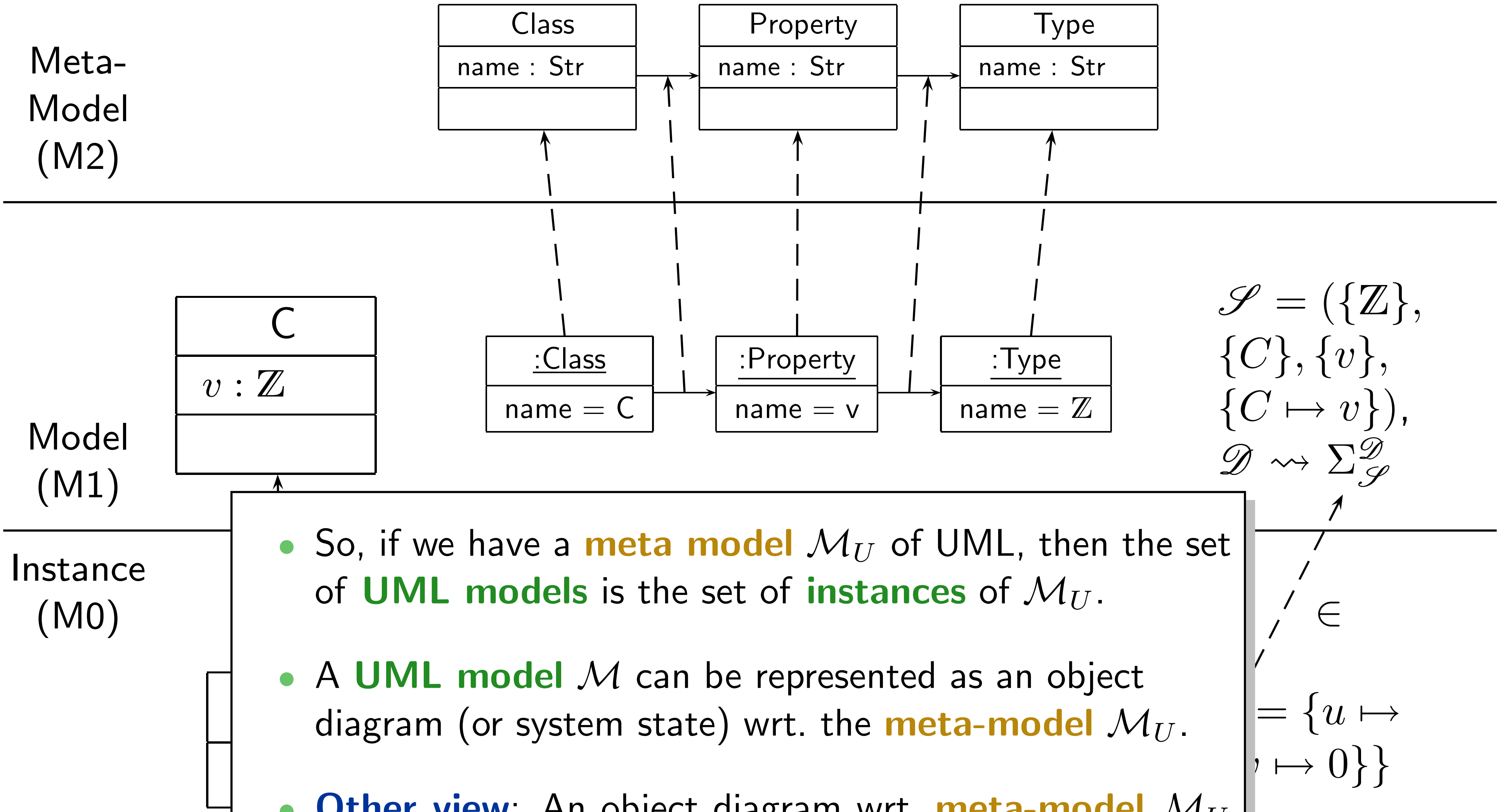
Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

Meta-Modelling: Principle

Modelling vs. Meta-Modelling



Modelling vs. Meta-Modelling



- So, if we have a **meta model** $\mathcal{M}_U$ of UML, then the set of **UML models** is the set of **instances** of $\mathcal{M}_U$.

- A **UML model** $\mathcal{M}$ can be represented as an object diagram (or system state) wrt. the **meta-model** $\mathcal{M}_U$.

- **Other view**: An object diagram wrt. **meta-model** $\mathcal{M}_U$ can (alternatively) be rendered as the **UML model** $\mathcal{M}$.

Well-Formedness as Constraints in the Meta-Model

- The set of **well-formed UML models** can be defined as the set of object diagrams satisfying all constraints of the **meta-model**.

  For example,

  > "[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.
  >
  > not self . allParents() -> includes(self)" [OMG, 2007b, 53]

- The other way round:

  Given a **UML model** $\mathcal{M}$, unfold it into an object diagram O_1 wrt. $\mathcal{M}_U$.
  If O_1 is a **valid** object diagram of $\mathcal{M}_U$ (i.e. satisfies all invariants from *Inv*$(\mathcal{M}_U)$), then $\mathcal{M}$ is a well-formed UML model.

  That is, if we have an object diagram **validity checker** for of the meta-modelling language, then we have a **well-formedness checker** for UML models.

Reading the Standard

Table of Contents

Reading the Standard

Table of Contents

Reading the Standard

Table of Contents

Reading the Standard Cont'd

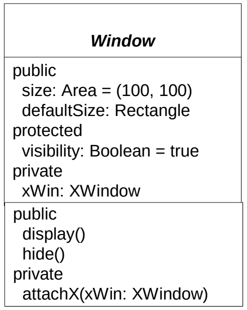


Figure 7.29 - Class notation: attributes and operations grouped according to visibility

7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

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

Generalizations

- "Namespace (from Kernel)" on page 99
- "RedefinableElement (from Kernel)" on page 130
- "Type (from Kernel)" on page 135

Description

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

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

A classifier is a redefinable element, meaning that it is possible to redefine nested classifiers.

Attributes

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

Associations

- /attribute: Property [*]
  Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets *Classifier::feature* and is a derived union.
- / feature : Feature [*]
  Specifies each feature defined in the classifier. Subsets *Namespace::member*. This is a derived union.
- / general : Classifier[*]
  Specifies the general Classifiers for this Classifier. This is derived.

Reading the Standard Cont'd

Wind

public
size: Area = (1
defaultSize: R
protected
visibility: Boole
private
xWin: XWindo
public
display()
hide()
private
attachX(xWin:

Figure 7.29 - Cl

7.3.8 Class

A classifier is a

Generalizatio

- "Namesp
- "Redefin
- "Type (fr

Description

A classifier is a

A classifier is a
other classifiers

A classifier is a

Attributes

- isAbstract:
  If *true,*
  classifi
  relation

Associations

- /attribute: P
  Refers
  Classif
- / feature : F
  Specifi
- / general : C
  Specifi

- generalization: Generalization[*]
  Specifies the Generalization relationships for this Classifier. These Generalizations navigate to more general classifiers in the generalization hierarchy. Subsets *Element::ownedElement*
- / inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers. Subsets *Namespace::member*. This is derived.
- redefinedClassifier: Classifier [*]
  References the Classifiers that are redefined by this Classifier. Subsets *RedefinableElement::redefinedElement*

Package Dependencies

- substitution : Substitution
  References the substitutions that are owned by this Classifier. Subsets *Element::ownedElement* and *NamedElement::clientDependency.*)

Package PowerTypes

- powertypeExtent : GeneralizationSet
  Designates the GeneralizationSet of which the associated Classifier is a power type.

Constraints

[1] The general classifiers are the classifiers referenced by the generalization relationships.

general = self.parents()

[2] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

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

[3] A classifier may only specialize classifiers of a valid type.

self.parents()->forAll(c | self.maySpecializeType(c))

[4] The inheritedMember association is derived by inheriting the inheritable members of the parents.

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

Package PowerTypes

[5] The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

Additional Operations

[1] The query allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as inheritance, this will be a larger set than feature.

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

allFeatures = member->select(oclIsKindOf(Feature))

[2] The query parents() gives all of the immediate ancestors of a generalized Classifier.

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

parents = generalization.general

Reading the Standard Cont'd

Wind

public
size: Area = (1
defaultSize: R
protected
visibility: Boole
private
xWin: XWindo

public
display()
hide()
private
attachX(xWin:

Figure 7.29 - Cl

7.3.8 Class

A classifier is a

Generalization

- "Namesp
- "Redefin
- "Type (fr

Description

A classifier is a

A classifier is a
other classifiers

A classifier is a

Attributes

- isAbstract: I
  If *true*,
  classifi
  relation

Associations

- /attribute: P
  Refers
  Classifi
- / feature : F
  Specifi
- / general : C
  Specifi

- generalizatio
  Specifi
  classifi
- / inheritedM
  Specifi
  derived
- redefinedCl
  Referen

Package Depe

- substitution
  Referen
  NamedE

Package Powe

- powertypeE
  Designa

Constraints

[1] The general
general = se

[2] Generalizati
transitively
not self.allP

[3] A classifier
self.parents(

[4] The inherite
self.inherited

Package Powe

[5] The Classifi
Generalizati
itself nor ma

Additional Op

[1] The query a
inheritance,
Classifier::a
allFeatures

[2] The query p
Classifier::p
parents = ge

[3] The query allParents() gives all of the direct and indirect ancestors of a generalized Classifier.

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

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

[4] The query inheritableMembers() gives all of the members of a classifier that may be inherited in one of its descendants, subject to whatever visibility restrictions apply.

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

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

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

[5] The query hasVisibilityOf() determines whether a named element is visible in the classifier. By default all are visible. It is only called when the argument is something owned by a parent.

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

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

if (self.inheritedMember->includes(n)) **then**
  hasVisibilityOf = (n.visibility <> #private)
else
  hasVisibilityOf = **true**

[6] The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example, in the specification of signature conformance for operations.

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

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

[7] The query inherit() defines how to inherit a set of elements. Here the operation is defined to inherit them all. It is intended to be redefined in circumstances where inheritance is affected by redefinition.

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

inherit = inhs

[8] The query maySpecializeType() determines whether this classifier may have a generalization relationship to classifiers of the specified type. By default a classifier may specialize classifiers of the same or a more general type. It is intended to be redefined by classifiers that have different specialization constraints.

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

maySpecializeType = self.oclIsKindOf(c.oclType)

Semantics

A classifier is a classification of instances according to their features.

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a classifier have values corresponding to the classifier's attributes.

A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.

Reading the S

Package PowerTypes

The notion of power type was inspired by the notion of power set. A power set is defined as a set whose instances are subsets. In essence, then, a power type is a class whose instances are subclasses. The powertypeExtent association relates a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets for that class.

Semantic Variation Points

The precise lifecycle semantics of aggregation is a semantic variation point.

Notation

Classifier is an abstract model element, and so properly speaking has no notation. It is nevertheless convenient to define in one place a default notation available for any concrete subclass of Classifier for which this notation is suitable. The default notation for a classifier is a solid-outline rectangle containing the classifier's name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier. The specific type of classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

The name of an abstract Classifier is shown in italics.

An attribute can be shown as a text string. The format of this string is specified in the Notation sub clause of "Property (from Kernel, AssociationClasses)" on page 123.

Presentation Options

Any compartment may be suppressed. A separator line is not drawn for a suppressed compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary.

An abstract Classifier can be shown using the keyword {abstract} after or below the name of the Classifier.

The type, visibility, default, multiplicity, property string may be suppressed from being displayed, even if there are values in the model.

The individual properties of an attribute can be shown in columns rather than as a continuous string.

Style Guidelines

- Attribute names typically begin with a lowercase letter. Multi-word names are often formed by concatenating the words and using lowercase for all letters except for upcasing the first letter of each word but the first.
- Center the name of the classifier in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above the classifier name.
- For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e, begin them with an uppercase character).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show full attributes and operations when needed and suppress them in other contexts or references.

Reading the S

Wind
public
size: Area = (1
defaultSize: R
protected
visibility: Boole
private
xWin: XWindo
public
display()
hide()
private
attachX(xWin:

Figure 7.29 - Cl

7.3.8 Class

A classifier is a

Generalization

- “Namesp
- “Redefin
- “Type (fr

Description

A classifier is a

A classifier is a other classifiers

A classifier is a

Attributes

- isAbstract: I If *true*, classifi relation

Associations

- /attribute: P Refers *Classifi*
- / feature : F Specifi
- / general : C Specifi

52

- generalizati Specifi classifi
- / inheritedM Specifi derived
- redefinedCl Referer

Package Depe

- substitution Referer *NamedI*

Package Powe

- powertypeE Design

Constraints

[1] The general general = se
[2] Generalizati transitively not self.allP
[3] A classifier self.parents
[4] The inherite self.inherited

Package Powe

[5] The Classifi Generalizati itself nor ma

Additional Op

[1] The query a inheritance, Classifier::a allFeatures
[2] The query p Classifier::p parents = ge

UML Superstructure Specification, v2.1.2 53

[3] The query a Classifier::a allParents =
[4] The query i subject to w Classifier::in pre: c.allPa inheritableM
[5] The query h only called Classifier::h pre: self.allP if (self.i ha else ha
[6] The query c in the specif Classifier::c conformsTo
[7] The query i to be redefi Classifier::in inherit = inh
[8] The query m the specifie redefined by Classifier::m maySpecial

Semantics

A classifier is a

A Classifier ma also an (indirec classifier are im general classifie

The specific sem classifier have v

A Classifier def to itself and to a

54

Package Powe

The notion of p subsets. In essen a Classifier with for that class.

Semantic Vari

The precise life

Notation

Classifier is an in one place a d default notation compartments s classifier can be

The name of an

An attribute can (from Kernel, A

Presentation O

Any compartme suppressed, no i to remove ambi

An abstract Cla

The type, visibil in the model.

The individual p

Style Guidelin

- Attribute and using
- Center th
- Center ke
- For those with an u
- Left justi
- Begin att
- Show ful

UML Superstructure Specification, v2.1.2 55

Examples

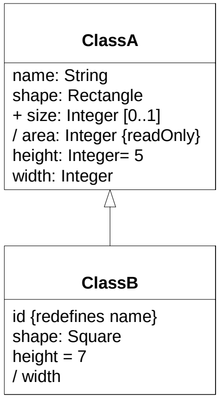


Figure 7.30 - Examples of attributes

The attributes in Figure 7.30 are explained below.

- ClassA::name is an attribute with type String.
- ClassA::shape is an attribute with type Rectangle.
- ClassA::size is a public attribute of type Integer with multiplicity 0..1.
- ClassA::area is a derived attribute with type Integer. It is marked as read-only.
- ClassA::height is an attribute of type Integer with a default initial value of 5.
- ClassA::width is an attribute of type Integer.
- ClassB::id is an attribute that redefines ClassA::name.
- ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.
- ClassB::height is an attribute that redefines ClassA::height. It has a default of 7 for ClassB instances that overrides the ClassA default of 5.
- ClassB::width is a derived attribute that redefines ClassA::width, which is not derived.

An attribute may also be shown using association notation, with no adornments at the tail of the arrow as shown in Figure 7.31.

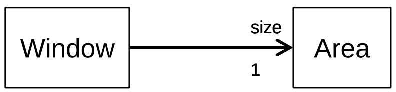


Figure 7.31 - Association-like notation for attribute

Reading the S

Package PowerTypes

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could then associate with two Generalizations where the class (i.e., general Classifier) Bank Account has two specific subclasses (i.e., Classifiers): Checking Account and Savings Account. Checking Account and Savings Account, then, are instances of the power type: Bank Account Type. In other words, Checking Account and Savings Account are *both:* instances of Bank Account Type, as well as subclasses of Bank Account. (For more explanation and examples, see Examples in the GeneralizationSet sub clause, below.)

7.3.9 Comment (from Kernel)

A comment is a textual annotation that can be attached to a set of elements.

Generalizations

- “Element (from Kernel)” on page 64.

Description

A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

A comment can be owned by any element.

Attributes

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

Associations

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

Constraints

No additional constraints

Semantics

A Comment adds no semantics to the annotated elements, but may represent information useful to the reader of the model.

Notation

A Comment is shown as a rectangle with the upper right corner bent (this is also known as a “note symbol”). The rectangle contains the body of the Comment. The connection to each annotated element is shown by a separate dashed line.

Presentation Options

The dashed line connecting the note to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagram.

Meta Object Facility (MOF)

Open Questions...

- Now you've been "**tricked**" again. Twice.
  - We didn't tell what the **modelling language** for meta-modelling is.
  - We didn't tell what the **is-instance-of** relation of this language is.
- **Idea**: have a **minimal object-oriented core** comprising the notions of **class**, **association**, **inheritance**, **etc.** with "self-explaining" semantics.
- This is **Meta Object Facility** (MOF), which (more or less) coincides with UML Infrastructure [OMG, 2007a].
- So: things on meta level
  - M0 are object diagrams/system states
  - M1 are **words of the language UML**
  - M2 are **words of the language MOF**
  - M3 are **words of the language** . . .

MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn't have a notion of Signal, our signature has.)

MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn't have a notion of Signal, our signature has.)
- Other approach:
  - Define a **generic**, **graph based** "is-instance-of" relation.
  - Object diagrams (that **are** graphs) then **are** the system states — not **only graphical representations** of system states.

MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn't have a notion of Signal, our signature has.)
- Other approach:
  - Define a **generic**, **graph based** "is-instance-of" relation.
  - Object diagrams (that **are** graphs) then **are** the system states — not **only graphical representations** of system states.
  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.

MOF Semantics

- One approach:
  - Treat it with **our signature-based theory**
  - This is (in effect) the right direction, but may require new (or extended) signatures for each level.
    (For instance, MOF doesn't have a notion of Signal, our signature has.)
- Other approach:
  - Define a **generic**, **graph based** "is-instance-of" relation.
  - Object diagrams (that **are** graphs) then **are** the system states — not **only graphical representations** of system states.
  - If this works out, good: We can easily experiment with different language designs, e.g. different flavours of UML that immediately have a semantics.
  - Most interesting: also do generic definition of behaviour within a closed modelling setting, but this is clearly still research, e.g. [?]

Meta-Modelling: (Anticipated) Benefits

Benefits: Overview

- We'll (superficially) look at three aspects:
  - Benefits for **Modelling Tools**.
  - Benefits for **Language Design**.
  - Benefits for **Code Generation and MDA**.

Benefits for Modelling Tools

- The meta-model $\mathcal{M}_U$ of UML **immediately** provides a **data-structure** representation for the abstract syntax ($\sim$ for our signatures).

  If we have code generation for UML models, e.g. into Java,
  then we can immediately represent UML models **in memory** for Java.

  (Because each MOF model is in particular a UML model.)

- There exist tools and libraries called **MOF-repositories**, which can generically represent instances of MOF instances (in particular UML models).

  And which can often generate specific code to manipulate instances of MOF instances in terms of the MOF instance.

Benefits for Modelling Tools Cont'd

- And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.

  $\rightarrow$ XML Metadata Interchange (XMI)

Benefits for Modelling Tools Cont'd

- And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.

  $\rightarrow$ XML Metadata Interchange (XMI)

- **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) $\rightarrow$ OMG Diagram Interchange.

Benefits for Modelling Tools Cont'd

- And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.

  $\rightarrow$ XML Metadata Interchange (XMI)

- **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) $\rightarrow$ OMG Diagram Interchange.

- **Note**: There are slight ambiguities in the XMI standard.

  And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.

  In some cases, it's possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.

  Plus XMI compatibility doesn't necessarily refer to Diagram Interchange.

Benefits for Modelling Tools Cont'd

- And not only **in memory**, if we can represent MOF instances in files, we obtain a canonical representation of UML models **in files**, e.g. in XML.

  $\rightarrow$ XML Metadata Interchange (XMI)

- **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) $\rightarrow$ OMG Diagram Interchange.

- **Note**: There are slight ambiguities in the XMI standard.

  And different tools by different vendors often seem to lie at opposite ends on the scale of interpretation. Which is surely a coincidence.

  In some cases, it's possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.

  Plus XMI compatibility doesn't necessarily refer to Diagram Interchange.

- **To re-iterate**: this is **generic for all** MOF-based modelling languages such as UML, CWM, etc.
  And also for **Domain Specific Languages** which don't even exit yet.

Benefits for Language Design

- Recall: we said that code-generators are possible “readers” of stereotypes.
- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. **corresponding** to the stereotype.

One mechanism to define DSLs (based on UML, and “within” UML): **Profiles**.

Benefits for Language Design

- Recall: we said that code-generators are possible "readers" of stereotypes.
- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. **corresponding** to the stereotype.

Et voilà: we can model Gtk-GUIs and generate code for them.

One mechanism to define DSLs (based on UML, and "within" UML): **Profiles**.

Benefits for Language Design

- Recall: we said that code-generators are possible "readers" of stereotypes.
- For example, (heavily simplifying) we could
  - introduce the stereotypes **Button**, **Toolbar**, ...
  - for convenience, instruct the modelling tool to use special pictures for stereotypes — in the meta-data (the abstract syntax), the stereotypes are clearly present.
  - instruct the code-generator to automatically add inheritance from Gtk::Button, Gtk::Toolbar, etc. **corresponding** to the stereotype.

  Et voilà: we can model Gtk-GUIs and generate code for them.
- Another view:
  - UML with these stereotypes **is a new modelling language**: Gtk-UML.
  - Which lives on the same meta-level as UML (M2).
  - It's a **Domain Specific** Modelling **Language** (DSL).

One mechanism to define DSLs (based on UML, and "within" UML): **Profiles**.

Benefits for Language Design Cont'd

- For each DSL defined by a Profile, we immediately have
  - in memory representations,
  - modelling tools,
  - file representations.

- **Note**: here, the **semantics** of the stereotypes (and thus the language of Gtk-UML) **lies in the code-generator**.

  That's the first "reader" that understands these special stereotypes. (And that's what's meant in the standard when they're talking about giving stereotypes semantics).

- One can also impose additional well-formedness rules, for instance that certain components shall all implement a certain interface (and thus have certain methods available). (Cf. [Stahl and Völter, 2005].)

Benefits for Language Design Cont'd

- One step further:
  - Nobody hinders us to obtain a model of UML (written in MOF),
  - throw out parts unnecessary for our purposes,
  - add (= integrate into the existing hierarchy) more adequat new constructs, for instance, **contracts** or something more close to hardware as **interrupt** or **sensor** or **driver**,
  - and maybe also stereotypes.

  $\rightarrow$ a new language standing next to UML, CWM, etc.

- Drawback: the resulting language is not necessarily UML any more, so we **can't use** proven UML modelling tools.

- But we can use all tools for MOF (or MOF-like things).

  For instance, Eclipse EMF/GMF/GEF.

Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for **re-factorings**, like moving common attributes upwards the inheritance hierarchy.

    This can now be defined as **graph-rewriting** rules on the level of MOF.
    The graph to be rewritten is the UML model

Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for **re-factorings**, like moving common attributes upwards the inheritance hierarchy.

    This can now be defined as **graph-rewriting** rules on the level of MOF.
    The graph to be rewritten is the UML model
  - Similarly, one could transform a **Gtk-UML** model into a **UML model**, where the inheritance from classes like Gtk::Button is made explicit:

    The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.

Benefits for Model (to Model) Transformation

- There are manifold applications for model-to-model transformations:
  - For instance, tool support for **re-factorings**, like moving common attributes upwards the inheritance hierarchy.

    This can now be defined as **graph-rewriting** rules on the level of MOF.
    The graph to be rewritten is the UML model
  - Similarly, one could transform a **Gtk-UML** model into a **UML model**, where the inheritance from classes like Gtk::Button is made explicit:

    The transformation would add this class Gtk::Button and the inheritance relation and remove the stereotype.
  - Similarly, one could have a **GUI-UML** model transformed into a **Gtk-UML** model, or a Qt-UML model.

    The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.

Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model.

  So code-generation is a **special case** of model-to-model transformation; only the destination looks quite different.

Special Case: Code Generation

- Recall that we said that, e.g. Java code, can also be seen as a model.

  So code-generation is a **special case** of model-to-model transformation; only the destination looks quite different.

- **Note**: Code generation needn't be as expensive as buying a modelling tool with full fledged code generation.

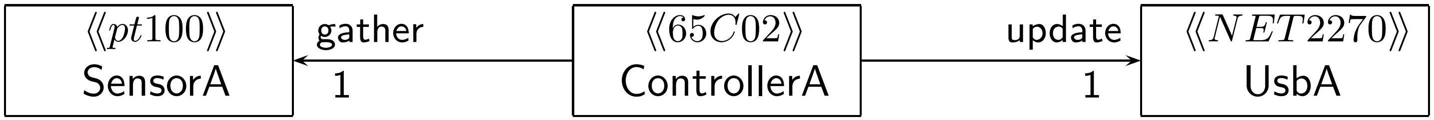
  - If we have the UML model (or the DSL model) given as an XML file, code generation can be **as simple as** an XSLT script.

    "Can be" in the sense of

    > *"There may be situation where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation."*

In general, code generation can (in colloquial terms) become **arbitrarily difficult**.

Example: Model and XMI




```
<?xml version = ’1.0’ encoding = ’UTF-8’ ?>
<XMI xmi.version = ’1.2’ xmlns:UML = ’org.omg.xmi.namespace.UML’ timestamp = ’Mon Feb 02 18:23:12 CET 2009’>
  <XMI.content>
    <UML:Model xmi.id = ’...’>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = ’...’ name = ’SensorA’>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = ’pt100’/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id = ’...’ name = ’ControllerA’>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = ’65C02’/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Class xmi.id = ’...’ name = ’UsbA’>
          <UML:ModelElement.stereotype>
            <UML:Stereotype name = ’NET2270’/>
          </UML:ModelElement.stereotype>
        </UML:Class>
        <UML:Association xmi.id = ’...’ name = ’in’ >...</UML:Association>
        <UML:Association xmi.id = ’...’ name = ’out’ >...</UML:Association>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

References

References

[Fischer and Wehrheim, 2000] Fischer, C. and Wehrheim, H. (2000). Behavioural subtyping relations for object-oriented formalisms. In Rus, T., editor, AMAST, number 1816 in Lecture Notes in Computer Science. Springer-Verlag.

[Liskov, 1988] Liskov, B. (1988). Data abstraction and hierarchy. SIGPLAN Not., 23(5):17–34.

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

[OMG, 2007b] OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

[Stahl and Völter, 2005] Stahl, T. and Völter, M. (2005). Modellgetriebene Softwareentwicklung. dpunkt.verlag, Heidelberg.