Software Design, Modelling and Analysis in UML

Lecture 21: Inheritance II

2014-02-05 10

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

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

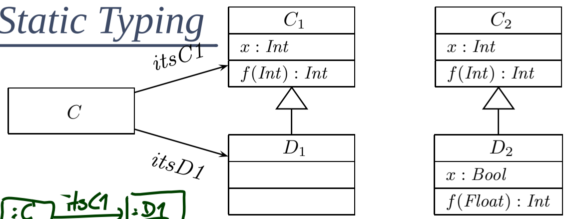
- Behavioural Features
- State Machines Variation Points
- Inheritance in UML: concrete syntax
- Liskov Substitution Principle — desired semantics

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:**
  - Two approaches to obtain desired semantics
  - The UML Meta Model

“...shall be usable...” for UML

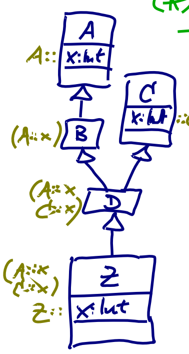
Easy: Static Typing



Given:

Wanted:

- $x > 0$ also **well-typed** for D_1
- assignment $itsC1 := itsD1$ being **well-typed** (∗)
- $itsC1.x = 0$, $itsC1.f(0)$, $itsC1 \,!\, F$
  being well-typed (and doing the right thing).

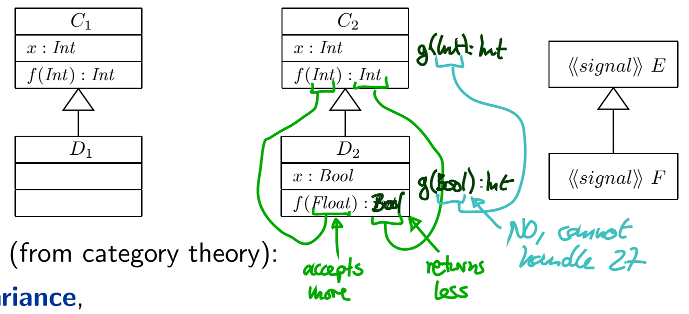


Approach:

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

Static Typing Cont'd

Assuming $\mathscr{D}(Bool) \subseteq \mathscr{D}(Int) \subseteq \mathscr{D}(Floats)$ plus corresponding type-system



Notions (from category theory):

- **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**).

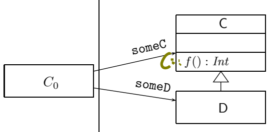
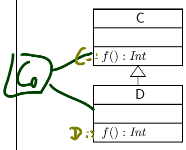
This is a notion used by many programming languages — and easily type-checked.

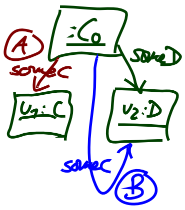
Excursus: Late Binding of Behavioural Features

Late Binding

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

the type of the link determines which impl. is used at runtime (not what the object really "is")

	f not overridden in D 	*f* overridden in D 	value of someC/ someD
`someC -> f()`	C::f()	C::f()	(A)
`someD -> f()`	C::f()	D::f()	(A)
`someC -> f()`	C::f()	C::f()	(B)



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

type of object determines which impl. is used

`someC -> f()`	C::f()	C::f()	(A)
`someD -> f()`	C::f()	D::f()	(A)
`someC -> f()`	C::f()	D::f()	(B)

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.)

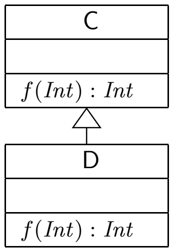
Back to the Main Track: "...tell the difference..." for UML

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 $C \preceq D$
  - via a C-link,
  - 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~~ Binding



- $C::f$ 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;
};
```

```
int C::f(int) {
   return (rand() % 2);
};
```

vs.

```
int D::f(int x) {
   return (x % 2);
};
```


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**).

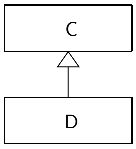
  Note: ~~This~~ (ii) is no longer a matter of simple type-checking!

- 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

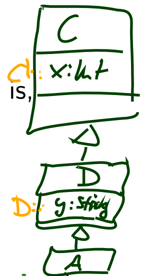


- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.
- But the state machine of a sub-class **cannot** be drawn from scratch.
- Instead, the state machine of a sub-class can only be obtained by applying actions from a **restricted** set to a copy of the 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



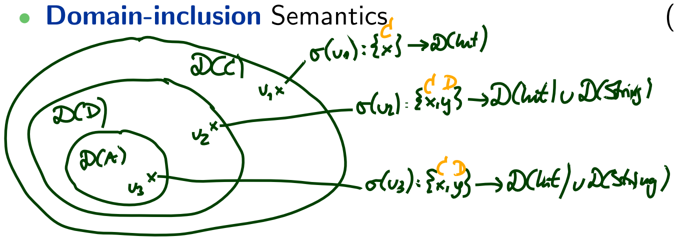
Wanted: a formal representation of "if $C \preceq D$ then D '**is a**' C", that is,

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

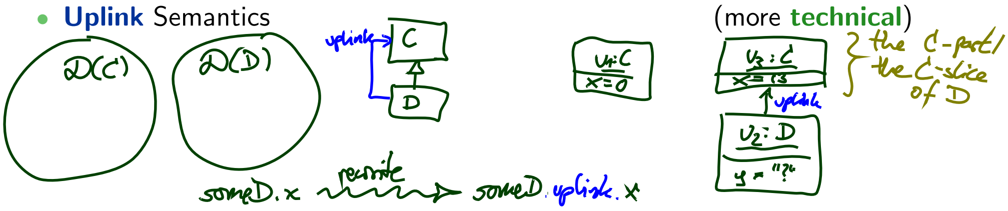
(ii) D objects (identities) can replace C objects.

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

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



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



Domain Inclusion Semantics

Domain Inclusion Structure

Let $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$ be a signature.

Now a **structure** $\mathscr{D}$

- [**as before**] maps types, classes, associations to domains,
- [**for completeness**] methods to transformers,
- [**as before**] indentities of instances of classes not (transitively) related by generalisation are disjoint,
- [**changed**] the indentities of a super-class comprise all identities of sub-classes, i.e.

$$\forall\, C \in \mathscr{C} : \mathscr{D}(C) \supsetneq \bigcup_{C \lhd D} \mathscr{D}(D).$$

Note: the old setting coincides with the special case $\lhd\ = \emptyset$.

Domain Inclusion System States

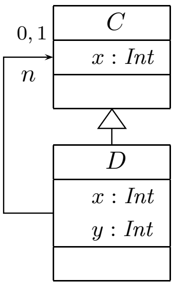
Now: a **system state** of $\mathscr{S}$ wrt. $\mathscr{D}$ is a **type-consistent** mapping

$$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_{0,1}) \cup \mathscr{D}(\mathscr{C}_*)))$$

that is, for all $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$,

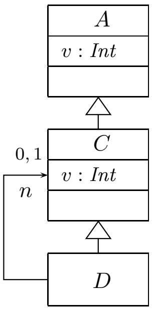
- [**as before**] $\sigma(u)(v) \in \mathscr{D}(\tau)$ if $v : \tau$, $\tau \in \mathscr{T}$ or $\tau \in \{C_*, C_{0,1}\}$.
- [**changed**] $\mathrm{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} atr(C_0)$,

Example:



Note: the old setting still coincides with the special case $\lhd\ = \emptyset$.

Preliminaries: Expression Normalisation



Recall:

- we want to allow, e.g., "context D inv : $v < 0$".
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, v shall denote the
"**most special more general**" $C::v$ according to $\lhd$.

To keep this out of typing rules, we assume that the following **normalisation** has been applied to all OCL expressions and all actions.

- Given expression v (or f) in **context** of class D, as determined by, e.g.
  - by the (type of the) navigation expression prefix, or
  - by the class, the state-machine where the action occcurs belongs to,
  - similar for method bodies,
- **normalise** v to (= replace by) $C::v$,
- where C is the **greatest** class wrt. "$\preceq$" such that
  - $C \preceq D$ and $C::v \in atr(C)$.

If no (unique) such class exists, the model is considered **not well-formed**; the expression is ambiguous. Then: explicitly provide the **qualified name**.

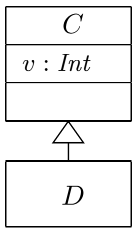
OCL Syntax and Typing

- Recall (part of the) OCL syntax and typing: $v, r \in V;\ C, D \in \mathscr{C}$

$$
\begin{aligned}
expr ::= \quad & v(expr_1) && : \tau_C \to \tau(v), && \text{if } v : \tau \in \mathscr{T} \\
\mid\ & r(expr_1) && : \tau_C \to \tau_D, && \text{if } r : D_{0,1} \\
\mid\ & r(expr_1) && : \tau_C \to Set(\tau_D), && \text{if } r : D_*
\end{aligned}
$$

The definition of the semantics remains (textually) **the same**.

More Interesting: Well-Typed-ness



- We want

$$\text{context } D \text{ inv} : v < 0$$

to be well-typed.

Currently it isn't because

$$v(expr_1) : \tau_C \to \tau(v)$$

but $A \vdash self : \tau_D$.

(Because τ_D and τ_C are still **different types**, although $\mathrm{dom}(\tau_D) \subset \mathrm{dom}(\tau_C)$.)

- So, add a (first) new typing rule

$$\frac{A \vdash expr : \tau_D}{A \vdash expr : \tau_C}, \text{ if } C \preceq D. \qquad \text{(Inh)}$$

Which is correct in the sense that, if '$expr$' is of type τ_D, then we can use it everywhere, where a τ_C is allowed.

The system state is prepared for that.

Well-Typed-ness with Visibility Cont'd

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = + \qquad \text{(Pub)}$$

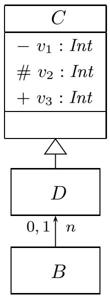
$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = \#, \; C \preceq D \qquad \text{(Prot)}$$

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = -, \; C = D \qquad \text{(Priv)}$$

$\langle C{::}v : \tau, \xi, v_0, P \rangle \in atr(C)$.

Example:

context/ inv	$(n.)v_1 < 0$	$(n.)v_2 < 0$	$(n.)v_3 < 0$
C			
D			
B			



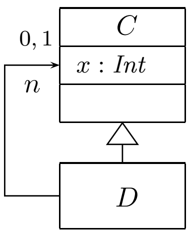
Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say $\mathcal{M} \models expr$ for $\underbrace{\text{context } C \text{ inv} : expr_0}_{=expr} \in Inv(\mathcal{M})$ iff

$$\forall \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!] \quad \forall i \in \mathbb{N} \quad \forall u \in \mathrm{dom}(\sigma_i) \cap \mathscr{D}(C) :$$
$$I[\![expr_0]\!](\sigma_i, \{self \mapsto u\}) = 1.$$

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $Inv(\mathcal{M})$.

- **Example**:



Transformers (Domain Inclusion)

- Transformers also remain **the same**, e.g. [VL 12, p. 18]

$$update(expr_1, v, expr_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon)$$

  with

$$\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma)]]$$

  where $u = I[\![expr_1]\!](\sigma)$.

Semantics of Method Calls

- **Non late-binding**: clear, by normalisation.
- **Late-binding**:
  Construct a **method call** transformer, which is applied to all method calls.

Inheritance and State Machines: Triggers

- **Wanted**: triggers shall also be sensitive for inherited events, sub-class shall execute super-class' state-machine (unless overridden).

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon') \text{ if}$$

- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)\ \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- a transition is enabled, i.e.

$$\exists\, (s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F = E \wedge I[\![expr]\!](\tilde{\sigma}) = 1$$

  where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]$.

and

- (σ', ε') results from applying t_{act} to (σ, ε) and removing u_E from the ether, i.e.

$$(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E),$$

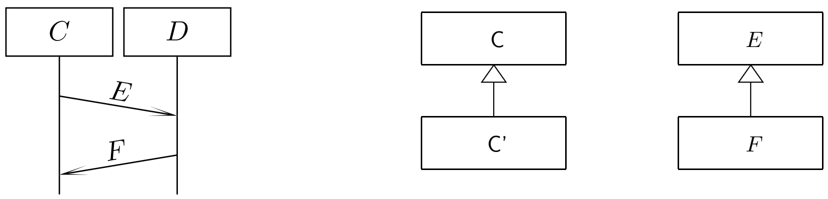
$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$$

  where b **depends**:
  - If u becomes stable in s', then $b = 1$. It **does** become stable if and only if there is no transition **without trigger** enabled for u in (σ', ε').
  - Otherwise $b = 0$.
- Consumption of u_E and the side effects of the action are observed, i.e.

$$cons = \{(u, (E, \sigma(u_E)))\}, Snd = Obs_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E).$$

– 21 – 2014-02-05 – Sdomincl –

Domain Inclusion and Interactions



- Similar to satisfaction of OCL expressions above:
  - An instance line stands for all instances of C (exact or inheriting).
  - Satisfaction of event observation has to take inheritance into account, too, so we have to **fix**, e.g.

$$\sigma, cons, Snd \models_\beta E^!_{x,y}$$

    if and only if

$$\beta(x) \text{ sends an } F\text{-event to } \beta y \text{ where } E \preceq F.$$

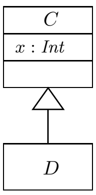
  - **Note**: C-instance line also binds to C'-objects.

– 21 – 2014-02-05 – Sdomincl –

Uplink Semantics

Uplink Semantics

- **Idea**:
  - Continue with the existing definition of **structure**, i.e. disjoint domains for identities.
  - Have an **implicit association** from the child to each parent part (similar to the implicit attribute for stability).



  - Apply (a different) pre-processing to make appropriate use of that association, e.g. rewrite (C++)

$$\mathtt{x} = 0;$$

in D to

$$\mathtt{uplink}_C \texttt{ -> } \mathtt{x} = 0;$$

Pre-Processing for the Uplink Semantics

- For each pair $C \lhd D$, extend D by a (fresh) association

$$uplink_C : C \text{ with } \mu = [1,1],\ \xi = +$$

(**Exercise**: public necessary?)

- Given expression v (or f) in the **context** of class D,
  - let C be the **smallest** class wrt. "$\preceq$" such that
    - $C \preceq D$, and
    - $C{::}v \in atr(D)$
  - then there exists (by definition) $C \lhd C_1 \lhd \ldots \lhd C_n \lhd D$,
  - **normalise** v to (= replace by)

$$uplink_{C_n} \texttt{->} \cdots \texttt{->}\, uplink_{C_1}.C{::}v$$

- Again: if no (unique) smallest class exists,
  the model is considered **not well-formed**; the expression is ambiguous.

Uplink Structure, System State, Typing

- Definition of structure remains **unchanged**.
- Definition of system state remains **unchanged**.
- Typing and transformers remain **unchanged** —
  the preprocessing has put everything in shape.

Satisfying OCL Constraints (Uplink)

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say

$$\mathcal{M} \models expr$$

for

$$\underbrace{\text{context } C \text{ inv} : expr_0}_{=expr} \in \mathit{Inv}(\mathcal{M})$$

if and only if

$$\begin{aligned} &\forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!] \\ &\quad \forall i \in \mathbb{N} \\ &\qquad \forall u \in \mathrm{dom}(\sigma_i) \cap \mathscr{D}(C) : \\ &\qquad\quad I[\![expr_0]\!](\sigma_i, \{self \mapsto u\}) = 1. \end{aligned}$$

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\mathit{Inv}(\mathcal{M})$.

Transformers (Uplink)

- What **has to change** is the **create** transformer:

$$create(C, expr, v)$$

- Assume, C's inheritance relations are as follows.

$$\begin{gathered} C_{1,1} \lhd \ldots \lhd C_{1,n_1} \lhd C, \\ \ldots \\ C_{m,1} \lhd \ldots \lhd C_{m,n_m} \lhd C. \end{gathered}$$

- Then, we have to
  - create one fresh object for each part, e.g.

$$u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m},$$

  - set up the uplinks recursively, e.g.

$$\sigma(u_{1,2})(uplink_{C_{1,1}}) = u_{1,1}.$$

  - And, if we had constructors, be careful with their order.

Late Binding (Uplink)

- Employ something similar to the "`mostspec`" trick (in a minute!). But the result is typically far from concise.

  (Related to OCL's `isKindOf`() function, and RTTI in C++.)

Domain Inclusion vs. Uplink Semantics

Cast-Transformers

- C c;
- D d;
- **Identity upcast** (C++):
  - C∗ cp = &d; *// assign address of 'd' to pointer 'cp'*
- **Identity downcast** (C++):
  - D∗ dp = (D∗)cp; *// assign address of 'd' to pointer 'dp'*
- **Value upcast** (C++):
  - ∗c = ∗d; *// copy attribute values of 'd' into 'c', or,*
    // more precise, the values of the C-part of 'd'

Casts in Domain Inclusion and Uplink Semantics

	Domain Inclusion	**Uplink**
C∗ cp = &d;	easy: immediately compatible (in underlying system state) because &d yields an identity from $\mathscr{D}(D) \subset \mathscr{D}(C)$.	easy: By pre-processing, C∗ cp = d.uplink$_C$;
D∗ dp = (D∗)cp;	easy: the value of cp is in $\mathscr{D}(D) \cap \mathscr{D}(C)$ because the pointed-to object is a D. Otherwise, error condition.	difficult: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)
c = d;	bit difficult: set (for all $C \preceq D$) $(C)(\cdot,\cdot) : \tau_D \times \Sigma \to \Sigma\vert_{atr(C)}$ $(u,\sigma) \mapsto \sigma(u)\vert_{atr(C)}$ Note: $\sigma' = \sigma[u_C \mapsto \sigma(u_D)]$ is not type-compatible!	easy: By pre-processing, c = ∗(d.uplink$_C$);

Identity Downcast with Uplink Semantics

- **Recall** (C++): `D d;` `C* cp = &d;` `D* dp = (D*)cp;`
- **Problem**: we need the identity of the D whose C-slice is denoted by cp.
- **One technical solution**:
  - Give up disjointness of domains for **one additional type** comprising all identities, i.e. have

$$\texttt{all} \in \mathscr{T}, \qquad \mathscr{D}(\texttt{all}) = \bigcup_{C \in \mathscr{C}} \mathscr{D}(C)$$

  - In each $\preceq$-**minimal class** have associations "`mostspec`" pointing to **most specialised** slices, plus information of which type that slice is.
  - Then **downcast** means, depending on the `mostspec` type (only finitely many possibilities), **going down and then up** as necessary, e.g.

```
switch(mostspec_type){
  case C :
    dp = cp -> mostspec -> uplink_{D_n} -> ... -> uplink_{D_1} -> uplink_D;
...
}
```


Domain Inclusion vs. Uplink Semantics: Differences

- **Note**: The uplink semantics views inheritance as an abbreviation:
  - We only need to touch transformers (create) — and if we had constructors, we didn't even needed that (we could encode the recursive construction of the upper slices by a transformation of the existing constructors.)
- **So**:
  - Inheritance **doesn't add** expressive power.
  - And it also **doesn't improve** conciseness **soo dramatically**.

As long as we're "**early binding**", that is...

Domain Inclusion vs. Uplink Semantics: Motives

- **Exercise**:

  What's the point of

  - having the **tedious** adjustments of the **theory** if it can be approached **technically**?
  - having the **tedious** technical **pre-processing** if it can be approached **cleanly** in the **theory**?

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 $\mathcal{M}_U$

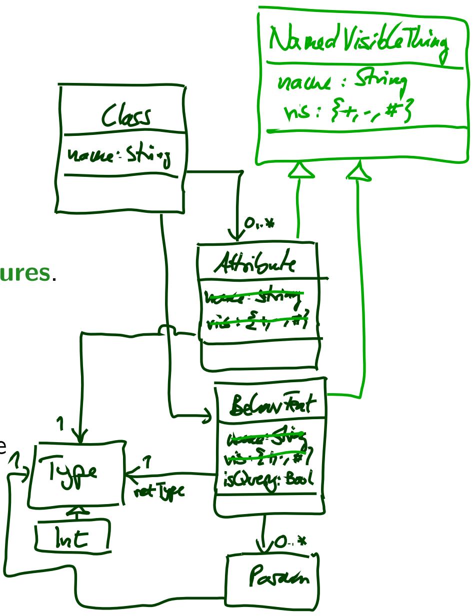
  is
  - the set of well-formed (!) UML models.

Meta-Modelling: Example

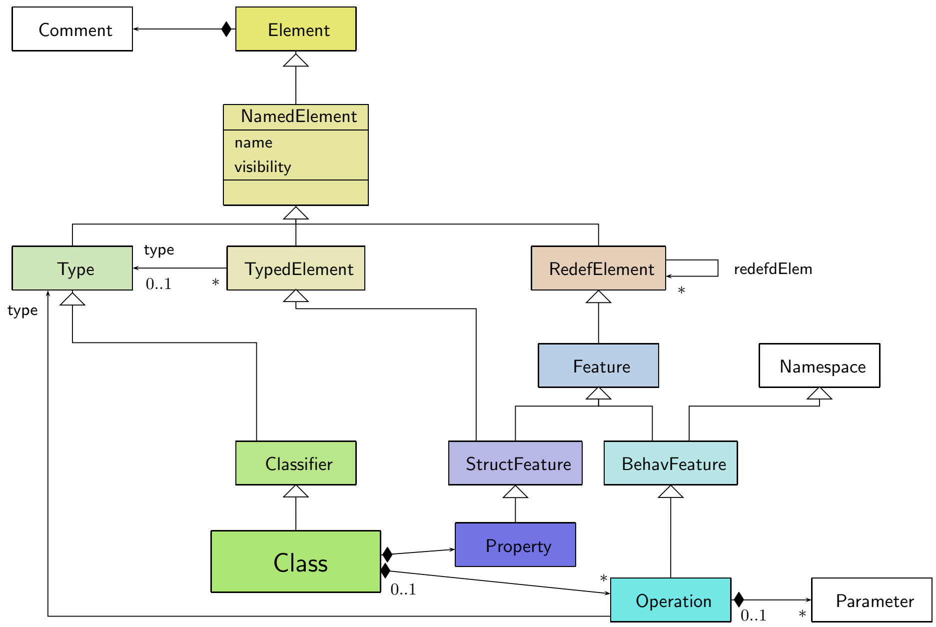
- For example, let's consider a class.
- A **class** has (on a superficial level)
  - a **name**,
  - any number of **attributes**,
  - any number of **behavioural features**.

  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]

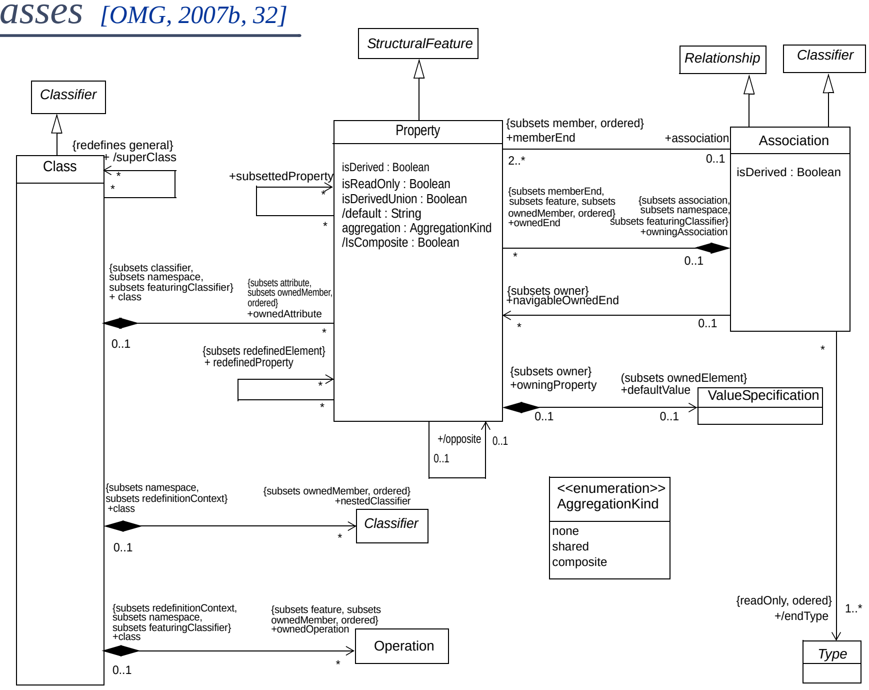


Figure 7.12 - Classes diagram of the Kernel package

Operations *[OMG, 2007b, 31]*

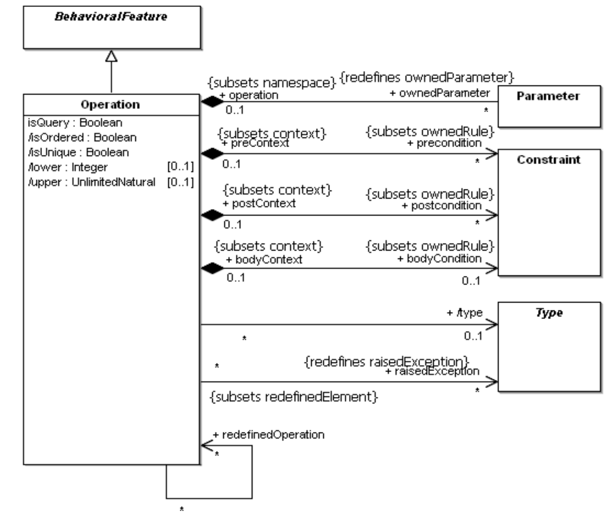


Figure 7.11 - Operations diagram of the Kernel package

Operations *[OMG, 2007b, 30]*

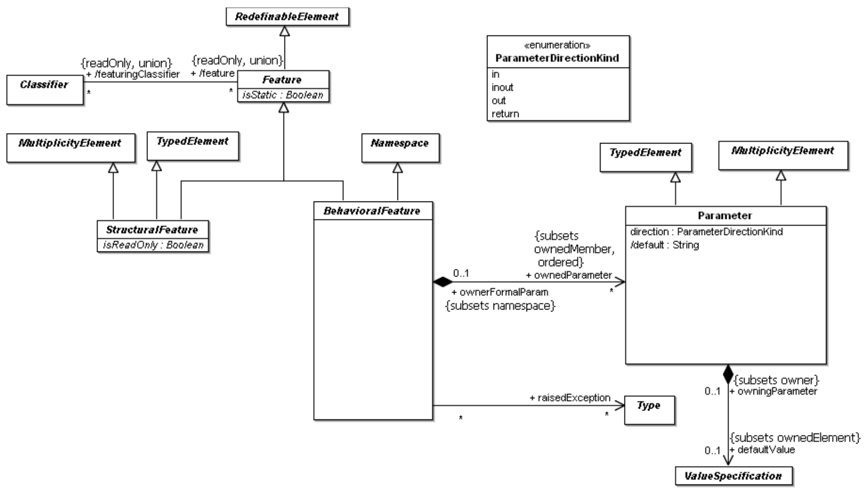


Figure 7.10 - Features diagram of the Kernel package

Classifiers [OMG, 2007b, 29]

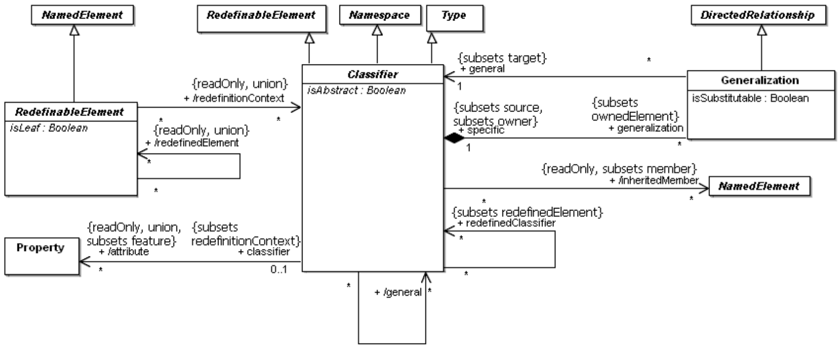


Figure 7.9 - Classifiers diagram of the Kernel package

Namespaces [OMG, 2007b, 26]

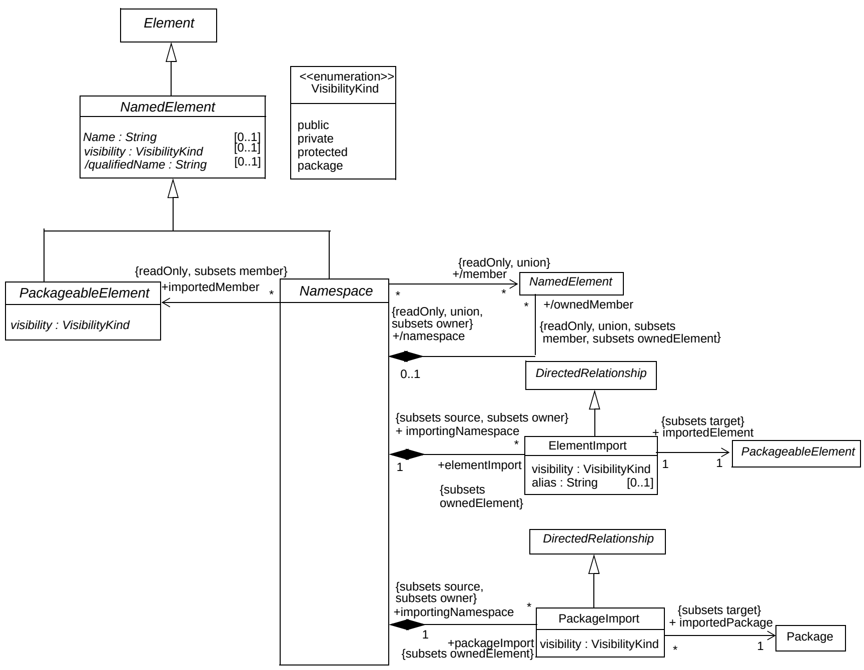


Figure 7.4 - Namespaces diagram of the Kernel package

Root Diagram *[OMG, 2007b, 25]*

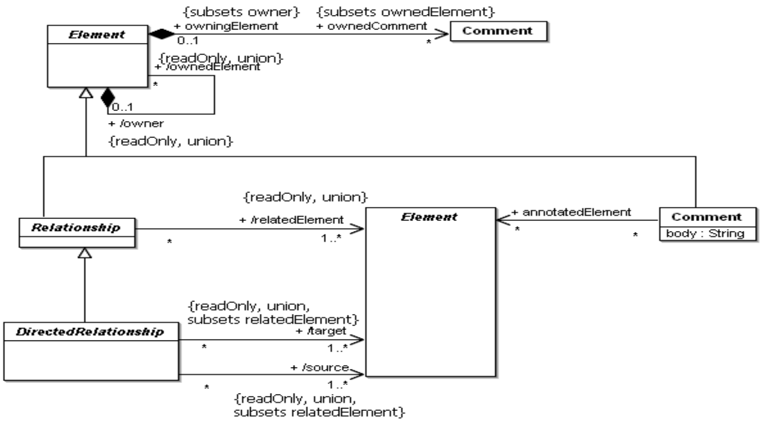


Figure 7.3 - Root diagram of the Kernel package

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

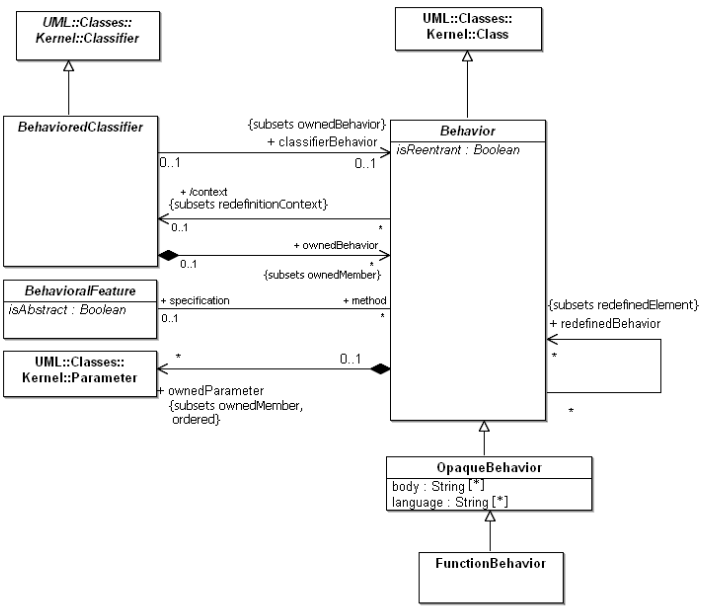


Figure 13.6 - Common Behavior

UML Architecture [OMG, 2003, 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.

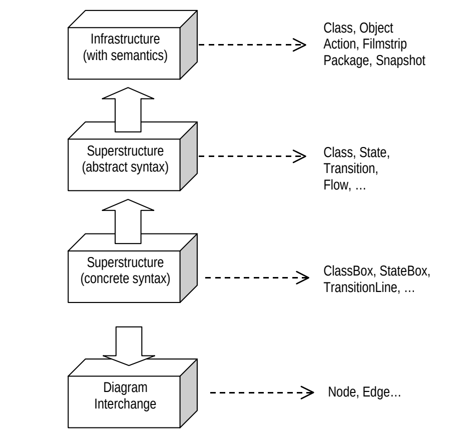
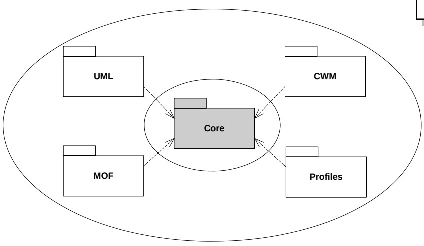


Figure 0-1 *Overview of architecture*



UML Superstructure Packages [OMG, 2007a, 15]

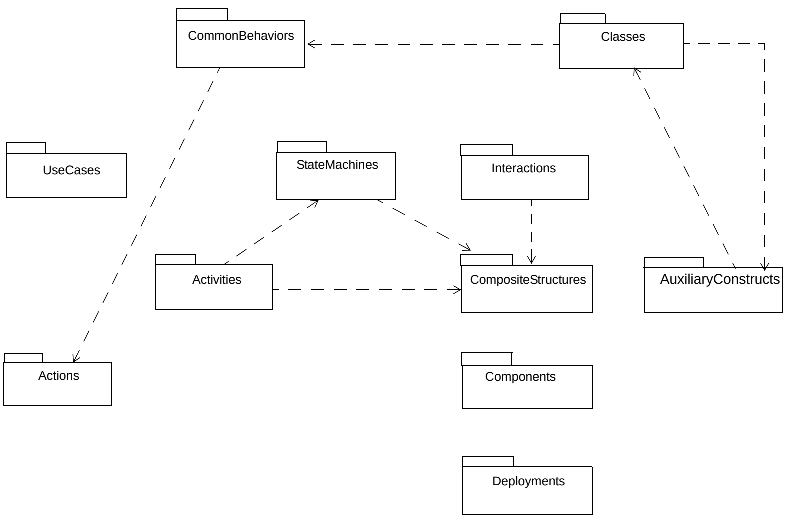
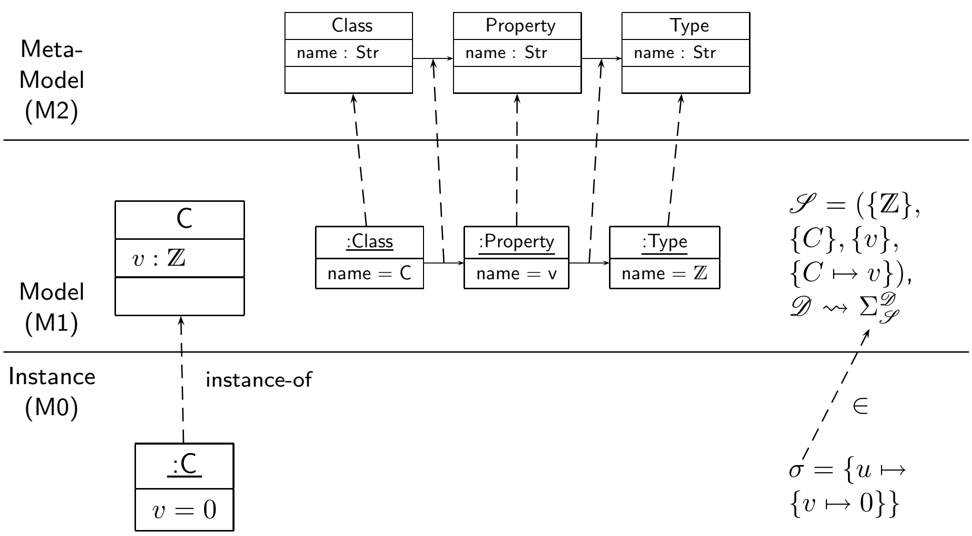


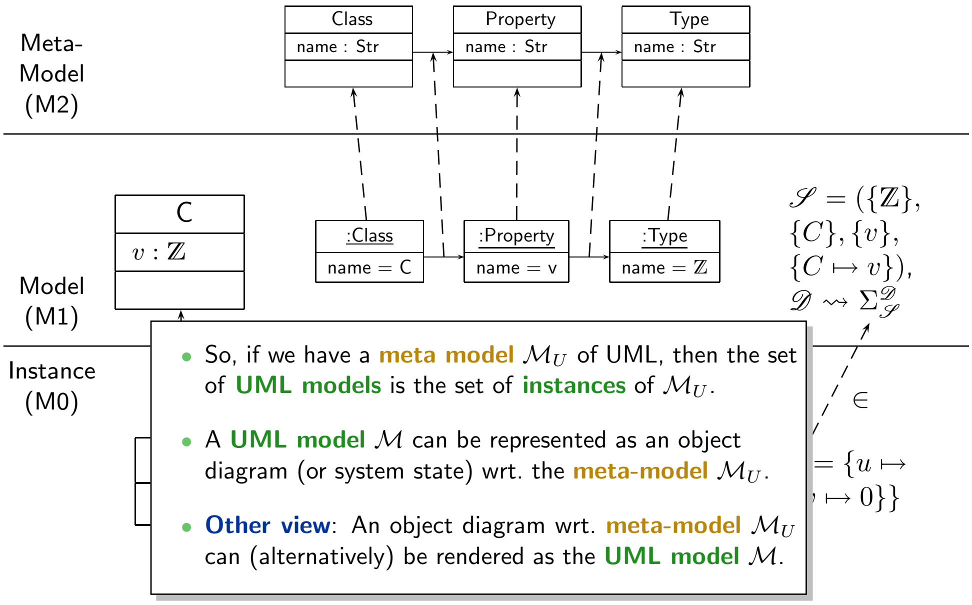
Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

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

1. Scope 1
2. Conformance 1
2.1 Language Units 2
2.2 Compliance Levels 2
2.3 Meaning and Types of Compliance 6
2.4 Compliance Level Contents 8
3. Normative References 10
4. Terms and Definitions 10
5. Symbols 10
6. Additional Information 10
6.1 Changes to Adopted OMG Specifications 10
6.2 Architectural Alignment and MDA Support 10
6.3 On the Run-Time Semantics of UML 11
6.3.1 The Basic Premises 11
6.3.2 The Semantics Architecture 11
6.3.3 The Basic Causality Model 12
6.3.4 Semantics Descriptions in the Specification 13
6.4 The UML Metamodel 13
6.4.1 Models and What They Model 13
6.4.2 Semantic Levels and Naming 14
6.5 How to Read this Specification 15
6.5.1 Specification format 15
6.5.2 Diagram format 18
6.6 Acknowledgements 19

Part I - Structure 21

7. Classes 23

Reading the Standard

Table of Contents

7.1 Overview 23
7.2 Abstract Syntax 24
7.3 Class Descriptions 38
7.3.1 Abstraction (from Dependencies) 38
7.3.2 AggregationKind (from Kernel) 38
7.3.3 Association (from Kernel) 39
7.3.4 AssociationClass (from AssociationClasses) 47
7.3.5 BehavioralFeature (from Kernel) 48
7.3.6 BehavioredClassifier (from Interfaces) 49
7.3.7 Class (from Kernel) 49
7.3.8 Classifier (from Kernel, Dependencies, PowerTypes) 52
7.3.9 Comment (from Kernel) 57
7.3.10 Constraint (from Kernel) 58
7.3.11 DataType (from Kernel) 60
7.3.12 Dependency (from Dependencies) 62
7.3.13 DirectedRelationship (from Kernel) 63
7.3.14 Element (from Kernel) 64
7.3.15 ElementImport (from Kernel) 65
7.3.16 Enumeration (from Kernel) 67
7.3.17 EnumerationLiteral (from Kernel) 68
7.3.18 Expression (from Kernel) 69
7.3.19 Feature (from Kernel) 70
7.3.20 Generalization (from Kernel, PowerTypes) 71
7.3.21 GeneralizationSet (from PowerTypes) 75
7.3.22 InstanceSpecification (from Kernel) 82
7.3.23 InstanceValue (from Kernel) 85
7.3.24 Interface (from Interfaces) 86
7.3.25 InterfaceRealization (from Interfaces) 89
7.3.26 LiteralBoolean (from Kernel) 89
7.3.27 LiteralInteger (from Kernel) 90
7.3.28 LiteralNull (from Kernel) 91
7.3.29 LiteralSpecification (from Kernel) 92
7.3.30 LiteralString (from Kernel) 92
7.3.31 LiteralUnlimitedNatural (from Kernel) 93
7.3.32 MultiplicityElement (from Kernel) 94
7.3.33 NamedElement (from Kernel, Dependencies) 97
7.3.34 Namespace (from Kernel) 99
7.3.35 OpaqueExpression (from Kernel) 101
7.3.36 Operation (from Kernel, Interfaces) 103
7.3.37 Package (from Kernel) 107
7.3.38 PackageableElement (from Kernel) 109
7.3.39 PackageImport (from Kernel) 110
7.3.40 PackageMerge (from Kernel) 111
7.3.41 Parameter (from Kernel, AssociationClasses) 120
7.3.42 ParameterDirectionKind (from Kernel) 122
7.3.43 PrimitiveType (from Kernel) 122
7.3.44 Property (from Kernel, AssociationClasses) 123
7.3.45 Realization (from Dependencies) 129
7.3.46 RedefinableElement (from Kernel) 130

Reading the Standard

Table of Contents

1. Scope . . . .
2. Conformance . . . . . .
 2.1 Language Units . . .
 2.2 Compliance Levels .
 2.3 Meaning and Types
 2.4 Compliance Level Co
3. Normative References
4. Terms and Definitions
5. Symbols . . . .
6. Additional Information
 6.1 Changes to Adopted
 6.2 Architectural Alignme
 6.3 On the Run-Time Se
  6.3.1 The Basic Premis
  6.3.2 The Semantics Ar
  6.3.3 The Basic Causal
  6.3.4 Semantics Descr
 6.4 The UML Metamodel
  6.4.1 Models and What
  6.4.2 Semantic Levels a
 6.5 How to Read this Sp
  6.5.1 Specification form
  6.5.2 Diagram format ..
 6.6 Acknowledgements

Part I - Structure . .

7. Classes . . . .

7.1 Overview . . . .
7.2 Abstract Syntax . . .
7.3 Class Descriptions .
 7.3.1 Abstraction (from
 7.3.2 AggregationKind
 7.3.3 Association (from
 7.3.4 AssociationClass
 7.3.5 BehavioralFeatur
 7.3.6 BehavioredClassi
 7.3.7 Class (from Kerne
 7.3.8 Classifier (from K
 7.3.9 Comment (from K
 7.3.10 Constraint (from
 7.3.11 DataType (from
 7.3.12 Dependency (fro
 7.3.13 DirectedRelatior
 7.3.14 Element (from K
 7.3.15 ElementImport (
 7.3.16 Enumeration (fro
 7.3.17 EnumerationLite
 7.3.18 Expression (from
 7.3.19 Feature (from Ke
 7.3.20 Generalization (
 7.3.21 GeneralizationS
 7.3.22 InstanceSpecific
 7.3.23 InstanceValue (f
 7.3.24 Interface (from Ir
 7.3.25 InterfaceRealiza
 7.3.26 LiteralBoolean (f
 7.3.27 LiteralInteger (fr
 7.3.28 LiteralNull (from
 7.3.29 LiteralSpecificat
 7.3.30 LiteralString (fro
 7.3.31 LiteralUnlimited
 7.3.32 MultiplicityEleme
 7.3.33 NamedElement
 7.3.34 Namespace (fro
 7.3.35 OpaqueExpress
 7.3.36 Operation (from
 7.3.37 Package (from K
 7.3.38 PackageableEle
 7.3.39 PackageImport (
 7.3.40 PackageMerge (
 7.3.41 Parameter (from
 7.3.42 ParameterDirect
 7.3.43 PrimitiveType (fr
 7.3.44 Property (from K
 7.3.45 Realization (from
 7.3.46 RedefinableElen

7.3.47 Relationship (from Kernel) 132
7.3.48 Slot (from Kernel) 132
7.3.49 StructuralFeature (from Kernel) 133
7.3.50 Substitution (from Dependencies) 134
7.3.51 Type (from Kernel) 135
7.3.52 TypedElement (from Kernel) 136
7.3.53 Usage (from Dependencies) 137
7.3.54 ValueSpecification (from Kernel) 137
7.3.55 VisibilityKind (from Kernel) 139
7.4 Diagrams 140
8. Components 143
8.1 Overview 143
8.2 Abstract syntax 144
8.3 Class Descriptions 146
8.3.1 Component (from BasicComponents, PackagingComponents) 146
8.3.2 Connector (from BasicComponents) 154
8.3.3 ConnectorKind (from BasicComponents) 157
8.3.4 ComponentRealization (from BasicComponents) 157
8.4 Diagrams 159
9. Composite Structures 161
9.1 Overview 161
9.2 Abstract syntax 161
9.3 Class Descriptions 166
9.3.1 Class (from StructuredClasses) 166
9.3.2 Classifier (from Collaborations) 167
9.3.3 Collaboration (from Collaborations) 168
9.3.4 CollaborationUse (from Collaborations) 171
9.3.5 ConnectableElement (from InternalStructures) 174
9.3.6 Connector (from InternalStructures) 174
9.3.7 ConnectorEnd (from InternalStructures, Ports) 176
9.3.8 EncapsulatedClassifier (from Ports) 178
9.3.9 InvocationAction (from InvocationActions) 178
9.3.10 Parameter (from Collaborations) 179
9.3.11 Port (from Ports) 179
9.3.12 Property (from InternalStructures) 183
9.3.13 StructuredClassifier (from InternalStructures) 186
9.3.14 Trigger (from InvocationActions) 190
9.3.15 Variable (from StructuredActivities) 191
9.4 Diagrams 191
10. Deployments 193

Reading the Standard Cont'd

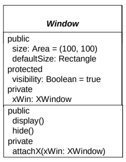


Figure 7.29 - Class notation: attributes and operations grouped according to visibility

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

- 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

[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.

– 21 – 2014-02-05 – Sreading –

57/74

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.

UML Superstructure Specification, v2.1.2 **55**

Reading the S

Examples

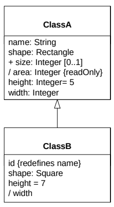


Figure 7.30 - Examples of attributes

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.

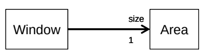
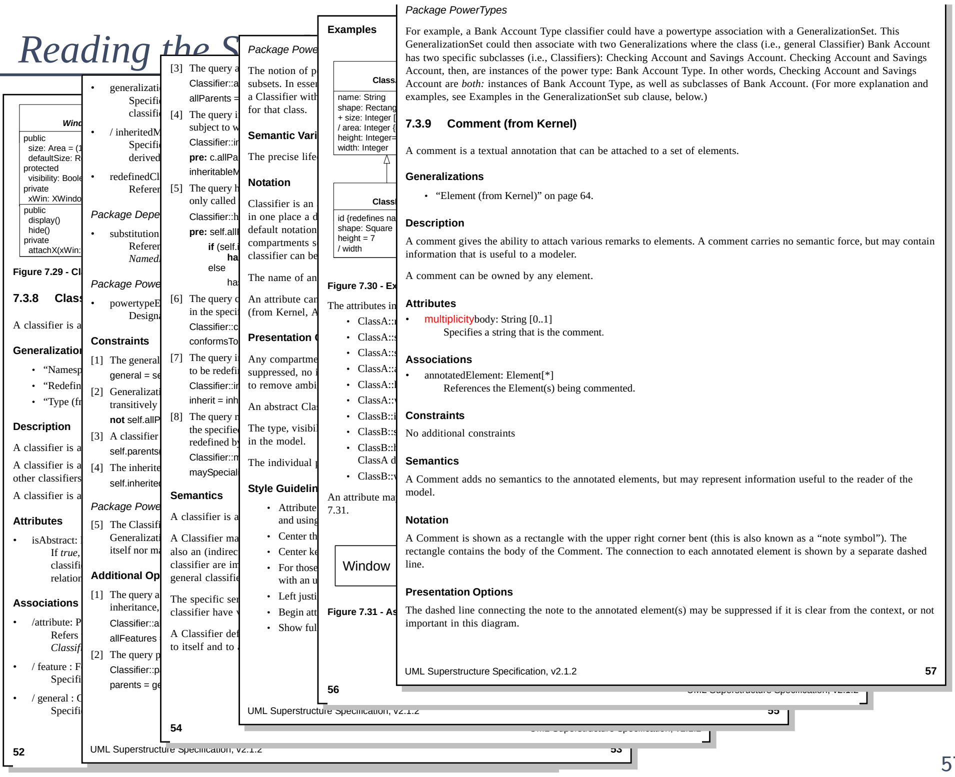


Figure 7.31 - Association-like notation for attribute

56 UML Superstructure Specification, v2.1.2

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.

UML Superstructure Specification, v2.1.2 57

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.)
- 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. [Buschermöhle and Oelerink, 2008]

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)

- **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: Overview

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

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.

  → 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: Overview

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

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.

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

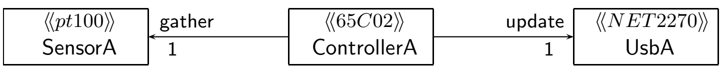
  - If we have the UML model (or the DSL model) given as an XML file, code generation can be **as simple as** an XSLT script.

    "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

[Buschermöhle and Oelerink, 2008] Buschermöhle, R. and Oelerink, J. (2008). Rich meta object facility. In Proc. 1st IEEE Int'l workshop UML and Formal Methods.

[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.

[OMG, 2003] OMG (2003). Uml 2.0 proposal of the 2U group, version 0.2, `http://www.2uworks.org/uml2submission`.

[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.