Software Design, Modelling and Analysis in UML

Lecture 21: Inheritance II

2014-02-05

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:

\[
\begin{align*}
C & \rightarrow \text{itsC1} \\
D_1 & \rightarrow \text{itsD1}
\end{align*}
\]

Wanted:

- \( x > 0 \) also well-typed for \( D_1 \)
- assignment \( \text{itsC1} := \text{itsD1} \) being well-typed (c)
- \( \text{itsC1}.x = 0, \text{itsC1}.f(0), \text{itsC1} ! F \) 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.

\[
\begin{align*}
\exp_1 := \exp_2 & \text{ is well-typed if} \\
\exp_1 : \tau_C & \text{ and } \exp_2 : \tau_D \text{ and } \tau_C \subseteq \tau_D
\end{align*}
\]
Static Typing Cont’d

\[ C_1 \xrightarrow{x: \text{Int}} f(\text{Int}) : \text{Int} \]
\[ D_1 \]

\[ C_2 \xrightarrow{x: \text{Int}} f(\text{Int}) : \text{Int} \]
\[ D_2 \xrightarrow{x: \text{Bool}} f(\text{Float}) : \text{Int} \]

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 \( \text{(contravariant)} \),
- provides a **more specialised** type as output \( \text{(covariant)} \).

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.)

<table>
<thead>
<tr>
<th>the type of the link</th>
<th>f not overridden in D</th>
<th>f overridden in D</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>the type of the link</th>
<th>f not overridden in D</th>
<th>f overridden in D</th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>C::f()</td>
<td>D::f()</td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td>C::f()</td>
</tr>
</tbody>
</table>

---

## 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.)
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 $C \leq D$
  - 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

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

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

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

vs.

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

```c
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 \( \mathcal{S} = (\mathcal{P}, \mathcal{C}, V, \text{atr}, \& F, \text{mth}, \triangleleft) \) be a signature.

Now a structure \( \mathcal{D} \)

- [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.

\[
\forall C \in \mathcal{C} : \mathcal{D}(C) \supseteq \bigcup_{C \triangleleft D} \mathcal{D}(D).
\]

Note: the old setting coincides with the special case \( \triangleleft = \emptyset \).
Domain Inclusion System States

Now: a system state of \( \mathcal{S} \) wrt. \( \mathcal{D} \) is a type-consistent mapping
\[
\sigma: \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{C}) \cup \mathcal{D}(\mathcal{C}_{0,1}) \cup \mathcal{D}(\mathcal{C}_*)))
\]
that is, for all \( u \in \text{dom}(\sigma) \cap \mathcal{D}(\mathcal{C}) \),
- [as before] \( \sigma(u)(v) \in \mathcal{D}(\tau) \) if \( v: \tau, \tau \in \mathcal{C} \) or \( \tau \in \{C_*, C_{0,1}\} \).
- [changed] \( \text{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} \text{atr}(C_0) \).

Example:

\[
\begin{array}{c}
\text{C} \\
x: \text{Int} \\
\hline \\
\text{D} \\
x: \text{Int} \\
y: \text{Int} \\
\hline \\
0,1 \\
\end{array}
\]

Note: the old setting still coincides with the special case \( \vartriangleright = \emptyset \).

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 \( \vartriangleright \).

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 occurs 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 \( C::v \in \text{atr}(C) \).

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: \( v, r \in V; C, D \in \mathcal{C} \)
  
  \[
  \text{expr ::= } v(\text{expr}_1) : \tau_C \rightarrow \tau(v), \quad \text{if } v : \tau \in \mathcal{P} \\
  \big| \, r(\text{expr}_1) : \tau_C \rightarrow \tau_D, \quad \text{if } r : D_{0,1} \\
  \big| \, r(\text{expr}_1) : \tau_C \rightarrow \text{Set}(\tau_D), \quad \text{if } r : D_*
  \]

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

More Interesting: Well-Typed-ness

- We want
  
  \[
  \text{context } D \quad \text{inv : } v < 0
  \]
  
  to be well-typed.

  Currently it isn’t because
  
  \[
  v(\text{expr}_1) : \tau_C \rightarrow \tau(v)
  \]

  but \( A \vdash \text{self} : \tau_D \).

  (Because \( \tau_D \) and \( \tau_C \) are still different types, although \( \text{dom}(\tau_D) \subset \text{dom}(\tau_C) \).)

- So, add a (first) new typing rule

  \[
  \frac{A \vdash \text{expr} : \tau_D}{A \vdash \text{expr} : \tau_C}, \quad \text{if } C \preceq D.
  \]

  \( \text{(Inh)} \)

  Which is correct in the sense that, if ‘expr’ is of type \( \tau_D \), then we can use it everywhere, where a \( \tau_C \) is allowed.

  The system state is prepared for that.
Well-Typed-ness with Visibility Cont’d

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

\[ \frac{A, D \vdash \text{expr} : \tau}{A, D \vdash C :: v(\text{expr}) : \tau}, \quad \xi = \# \quad C \preceq D \quad \text{(Prot)} \]

\[ \frac{A, D \vdash \text{expr} : \tau}{A, D \vdash C :: v(\text{expr}) : \tau}, \quad \xi = - \quad C = D \quad \text{(Priv)} \]

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

Example:

<table>
<thead>
<tr>
<th>context/inv</th>
<th>(n.)v1 &lt; 0</th>
<th>(n.)v2 &lt; 0</th>
<th>(n.)v3 &lt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
\text{context/inv} \\
\hline
C & (n.)v1 < 0 & (n.)v2 < 0 & (n.)v3 < 0 \\
D & \text{Int} & \# & + \\
B & 0, 1 & n \\
\end{array}
\]

Satisfying OCL Constraints (Domain Inclusion)

- Let \( M = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{M}, \mathcal{S}) \) be a UML model, and \( \mathcal{D} \) a structure.

- We (continue to) say \( M \models \text{expr} \) for context \( C \), \( \text{inv} : \text{expr}_0 \in \text{Inv}(M) \) iff

\[
\forall \pi = (\sigma, \varepsilon) \in M \quad \forall i \in \mathbb{N} \quad \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) : \quad I[\text{expr}_0](\sigma_i, \{\text{self} \mapsto u\}) = 1.
\]

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

Example:
Transformers (Domain Inclusion)

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

\[
\text{update}(\text{expr}_1, v, \text{expr}_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon)
\]

with

\[
\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\text{expr}_2](\sigma)]]
\]

where \(u = I[\text{expr}_1](\sigma)\).

Semantics of Method Calls

- Non late-binding: clear, by normalisation.
- Late-binding:
  Construct a method call transformer, which is applied to all method calls.
**Wanted:** triggers shall also be sensitive for inherited events, sub-class shall execute super-class’ state-machine (unless overridden).

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

- \(\exists u \in \text{dom}(\sigma) \cap \mathcal{P}(C) \exists u \in \mathcal{P}(\delta), u \in \text{ready}(\varepsilon, u)\)
- \(u\) is stable and in state machine state \(s\), i.e. \(\sigma(u)(\text{stable}) = 1\) and \(\sigma(u)(s) = s\)
- a transition is enabled, i.e.
  \[
  \exists (s, F, \text{expr}, \text{act}, s') \in \Delta \text{ of } \mathcal{SM} \text{ s.t. } F = E \land I[\text{expr}] \sigma = 1
  \]
  where \(\sigma = \sigma[u.\text{params} = E \mapsto u].\)

and

- \((\sigma', \varepsilon')\) results from applying \(t_{\text{act}}\) to \((\sigma, \varepsilon)\) and removing \(u_E\) from the ether, i.e.
  \[
  (\sigma'', \varepsilon') = t_{\text{act}}(\sigma', \varepsilon \oplus u_E),
  \]
  \[
  \sigma' = (\sigma''[u.\text{st} \mapsto s', u.\text{stable} \mapsto b, u.\text{params} = E \mapsto \emptyset])|_{(u \oplus u_E)}
  \]
  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 \((\sigma', \varepsilon').\)
  - Otherwise \(b = 0\).

- Consumption of \(u_E\) and the side effects of the action are observed, i.e.
  \[
  \text{cons} = \{(u, (E, \sigma(u_E)))\}, \text{Snd} = \text{Obs}_{\text{act}}(\sigma, \varepsilon \oplus u_E).
  \]

**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.
    \[
    \sigma, \text{cons}, \text{Snd} \models_\beta E_x, y
    \]
    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.
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).

```
C
x : Int

D
```

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

```
x = 0;
```

in $D$ to

```
uplink_C \rightarrow x = 0;
```
Pre-Processing for the Uplink Semantics

- For each pair $C \triangleleft D$, extend $D$ by a (fresh) association
  
  $$\text{uplink}_C : C \text{ with } \mu = [1, 1], \ \xi = +$$

  (Exercise: public necessary?)

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

- 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 \( \mathcal{M} = (\mathcal{D}, \mathcal{O}, \mathcal{I}, \mathcal{M}, \mathcal{I}) \) be a UML model, and \( \mathcal{D} \) a structure.
- We (continue to) say
  \[ \mathcal{M} \models expr \]
  for
  \[ \text{context } C \text{ inv : } expr_0 \in \text{inv}(\mathcal{M}) = expr \]
  if and only if
  \[ \forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in [\mathcal{M}] \]
  \[ \forall i \in \mathbb{N} \]
  \[ \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) : \]
  \[ I[[expr_0]()]_{\sigma_i, \{ \text{self } \mapsto u \}) = 1. \]
- \( \mathcal{M} \) is (still) consistent if and only if it satisfies all constraints in \( \text{inv}(\mathcal{M}) \).

Transformers (Uplink)

- What has to change is the create transformer:
  \[ \text{create}(C, \text{expr}, \nu) \]
- Assume, \( C \)'s inheritance relations are as follows.
  \[ C_{1,1} \subset \ldots \subset C_{1,n_1} \subset C, \]
  \[ \ldots \]
  \[ C_{m,1} \subset \ldots \subset C_{m,n_m} \subset C. \]
- 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.
    \[ \sigma(u_{1,2})(\text{uplink}_{C_{1,1}}) = u_{1,1}. \]
- And, if we had constructors, be careful with their order.
Late Binding (Uplink)

- Employ something similar to the “most spec” 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**

<table>
<thead>
<tr>
<th></th>
<th>Domain Inclusion</th>
<th>Uplink</th>
</tr>
</thead>
<tbody>
<tr>
<td>C* cp = &amp;d;</td>
<td>easy: immediately compatible (in underlying system state) because &amp;d yields an identity from ( \mathcal{P}(D) \subseteq \mathcal{P}(C) ).</td>
<td>easy: By pre-processing, C* cp = d.uplink(_C);</td>
</tr>
<tr>
<td>D* dp = (D*)cp;</td>
<td>easy: the value of cp is in ( \mathcal{P}(D) \cap \mathcal{P}(C) ) because the pointed-to object is a D. Otherwise, error condition.</td>
<td>difficult: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)</td>
</tr>
<tr>
<td>c = d;</td>
<td>bit difficult: set (for all ( C \subseteq D )) ( (C)(\cdot, \cdot) : \tau_D \times \Sigma \to \Sigma</td>
<td>_{\text{str}(C)} ) ( (u, \sigma) \mapsto \sigma(u)</td>
</tr>
</tbody>
</table>
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
    \[ \text{all} \in \mathcal{T}, \quad \mathcal{D} = \bigcup_{C \in \mathcal{E}} \mathcal{D}(C) \]
  - 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.
    ```cpp
    switch(mostspec_type){
    case C :
        dp = cp -> mostspec -> uplink_{Dn} -> ... -> uplink_{D1} -> 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 does not add expressive power.
    - And it also doesn’t improve conciseness so dramatically.

As long as we’re “early binding”, that is...
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 $M_U$ such that
  - the set of legal instances of $M_U$
  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**

![UML Meta-Model Diagram]

**Classes [OMG, 2007b, 32]**

![Classes Diagram]

Figure 7.12 - Classes diagram of the Kernel package
Figure 7.11 - Operations diagram of the Kernel package

Figure 7.10 - Features diagram of the Kernel package
Classifiers [OMG, 2007b, 29]

![Classifier diagram of the Kernel package]

Namespaces [OMG, 2007b, 26]

![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.

![Diagram](image_url)

**Figure 6-1 Overview of architecture**

**UML Superstructure Packages [OMG, 2007a, 15]**

![Diagram](image_url)

**Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure**
Meta-Modelling: Principle

Modelling vs. Meta-Modelling

Meta-Model (M2)

Model (M1)

Instance (M0)

\[ \mathcal{I} = \{\{Z\}, \{C\}, \{v\}, \{C \mapsto v\}\}, \sigma \rightsquigarrow \Sigma^{\mathcal{I}} \]

\[ \sigma = \{u \mapsto \{v \mapsto 0\}\} \]
Modelling vs. Meta-Modelling

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 $M$, unfold it into an object diagram $O_1$ wrt. $M_U$.
If $O_1$ is a valid object diagram of $M_U$ (i.e. satisfies all invariants from $Inv(M_U)$),
then $M$ 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.
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 Issues and What They Model ............... 13
      6.4.2 Dynamic Levels and Notation ............. 14
   6.5 How to Read this Specification .................... 15
      6.5.1 Specification format ..................... 15
      6.5.2 Diagram format ........................... 18
      6.5.3 Acknowledgements ....................... 19
Part I - Structure .................................... 21
7. Classes ................................................. 23

TABLE OF CONTENTS

Part I - Structure

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 Issues and What They Model ............... 13
      6.4.2 Dynamic Levels and Notation ............. 14
   6.5 How to Read this Specification .................... 15
      6.5.1 Specification format ..................... 15
      6.5.2 Diagram format ........................... 18
      6.5.3 Acknowledgements ....................... 19
Part I - Structure .................................... 21
7. Classes ................................................. 23
7.4 Diagrams

7.5 StructuralAbstraction *(from Kernel)*

7.5.1 Type *(from Kernel)*

7.5.2 RedefinedType *(from Kernel)*

7.5.3 Usage *(from Dependencies)*

7.5.4 Visibility *(from Kernel)*

7.5.5 ValueSpecification *(from Kernel)*

7.6 Generalization *(from Kernel)*

8. Components

8.1 Overview

8.2 Abstract syntax

8.3 Class Descriptions

8.4 Diagrams

9. Composite Structures

9.1 Overview

9.2 Abstract syntax

9.3 Class Descriptions

9.4 Diagrams

10. Deployments

Reading the Standard Cont'd
A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships. A classifier is a name space whose members can include features. A classifier is an abstract metaclass.

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

The query parents() gives all of the immediate ancestors of a generalized Classifier. In other words, a parent type may not be an instance of itself or any ancestor in its hierarchy.

Additional Operations

- pre: Classifier :: allParents()-> includesAll(pre)
- if: Classifier :: specializeType(c : Classifier) : Boolean;
- isAbstract: Boolean
- name: String
- namespace: Namespace
- no features: Feature
- ownedMember: OwnedMember
- operation: Operation
-owl: OWL

References for Clasifiers that are surrounded by the Classifier. Subtypes, Realizes, and Element are surrounded by a bold face font.

Package Dependencies

- subtyping: Subtyping
- references: References to the classifiers that are owned by the Classifier.
- superocl: Superocl

Figure 7.30 - Class notation: attributes and operations grouped according to visibility

The query inheritableMembers() gives all of the members of a classifier that may be inherited in one of its descendants, subject to visibility restrictions, if any. The query isAllInheritable Members() returns true if the argument is something owned by parent Classifier (hasValueOf() for AnonymousClassifier). Replaces allInheritableMembers().

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

References

- UML Superstructure Specification, v2.1.2

Figure 7.31 - Class notation: attributes and operations grouped according to visibility

The query allParents() returns the Classifier, immediately and transitively. The query inheritsFrom() determines whether the classifier may have a generalization relationship to classifiers of the specific type. The query inheritsFrom() is optional if the classifier is a superocl. It is intended to be overridden by classifiers that have different specialization conditions. A Classifier may participate in generalization relationships with other Classifier. An instance of a specific Classifier is also an 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 specific classifier also applies to instances of the specific classifier.
A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier is a namespace whose members can include features. A classifier is an abstract meta-class.

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

Generalizations
- /attribute: Property[*]
- /general: Classifier[*]

Visibility: Boolean = true
- public
- private

visibility = 'public' or 'private'

Figure 7.31 - Association-like notation for attribute

ClassA::area is a derived attribute with type Integer. It is marked as read-only.

ClassB::height is an attribute that redefines ClassA::height. It has a default of 7 for ClassB instances that overrides the default.

Summary:

- The name of an abstract classifier is shown in italic.
- The name of a classifier is shown in boldface.
- Center key (including stereotype names) in plain face within guillemets above the classifier name.
- With an uppercase character.

Presentation Options
- Any compartment may be suppressed. A separate line is not drawn for an suppressed compartment.
- If a compartment is suppressed, an inference cannot be drawn about the presence or absence of elements in it. Compartments suppressed must be shown in a continuous string.
- Style Guidelines
  - Attribute names typically begin with a lowercase letter. Mislabeled names are often formed by concatenating the results and upper transform for all letters except the first letter of each word, usually:
  - Class the name of the classifier to abbreviate.
  - Context keyword (including stereotype name), class line within parentheses after the classifier name.
  - For these languages that distinguish between uppercase and lowercase characters, capital names (e.g. begin them with an upper-case character).
  - If multiple attributes and operations are listed:
    - Begin attribute and operation names with a lowercase.
    - More attributes and operations (when needed and suppressed in columns) are shown in a continuous string.

Semantics
- The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a specific classifier must be specialized into instances of the specific classifier.

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is only called when the argument is something owned by a parent.

Additional Operations
- Classifier::parents(): Set(Classifier);
- Classifier::allParents(): Set(Classifier);
- Classifier::allParents = self.parents()->union(self.parents()->collect(p | p.allParents()));
- Classifier::inherits(inh : Set(NamedElement)): Set(NamedElement);
- Classifier::inherits(inh : Set(NamedElement));
- Classifier::conformsTo(other : Classifier): Boolean;
- Classifier::inherits(inh : Set(NamedElement));

Style Guide
- Attributes: an attribute is defined with type Integer.
- Class shape is a classifier with stereotype name.
- Class name is a public attribute of type Integer with multiplicity 0..1.
- Class name is a classifier attribute with type Integer. It is marked as read-only.
- Class height is an attribute of type Integer with a default initial value of 5.
- Class width is an attribute of type Integer.
- Class height is an attribute that ends with Class height. It has type Integer, a specialization of Integer.
- Class width is an attribute that ends with Class width. It is a default of 7.
- Class height must be specialized with a stereotype name.
- Class width is a derived attribute that ends with Class width, which is not derived.

Attributes may also be shown using association notation, with an adornment at the end of the arrow as shown in Figure 7.31.

Example

Figure 7.30 - Examples of attributes

- Class name is a attribute with type Integer.
- Class name is a classifier with stereotype name.
- Class name is a public attribute of type Integer with multiplicity 0..1.
- Class name is a classifier attribute with type Integer. It is marked as read-only.
- Class height is an attribute of type Integer with a default initial value of 5.
- Class width is an attribute of type Integer.
- Class height is an attribute that ends with Class height. It has type Integer, a specialization of Integer.
- Class width is an attribute that ends with Class width. It is a default of 7. The Class height must be specialized with a stereotype name.
- Class width is a derived attribute that ends with Class width, which is not derived.

Attributes may also be shown using association notation, with an adornment at the end of the arrow as shown in Figure 7.31.
A class ifi er is a n ame space whose members can include features. Classifier is a n abstract meta clas s.

A classifier is a red efinition eleme nt, meaning that it is possible to redefine nested classifiers.

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

References the Classifiers that are red efined by this Classifier. Subsets may special ize Type = self.oclIsKindOf(c.oclType)

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

• 
• 
•  

The notion of borders is used if "Classifier" is not for that class.

The notion of borders is used if "Classifier" is not for that class.

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

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 and annotated element is shown by a separate dashed line.

Style Guidelines

• Attributes
• Association ends
• Operations
• Features
• Generalizations

Figure 7.31-45

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

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 and 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 close from the context, or not important in this diagram.
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 adequate 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
<?xml version='1.0' encoding='UTF-8' ?>
<XMI xmi.version='1.2' xmlns:XMI='org.omg.xmi.namespace.XMI' 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:Namespace.ownedElement>
      <UML:Association xmi.id='...' name='in'>...</UML:Association>
      <UML:Association xmi.id='...' name='out'>...</UML:Association>
    </UML:Model>
  </XMI.content>
</XMI>
```
References


