

# Software Design, Modelling and Analysis in UML

## Lecture 21: Inheritance II

2014-02-05

Prof. Dr. Andreas Podolski, 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 LSCG, State Machines, System States?
  - What's the idea of Meta-Modeling?
- **Content:**
  - Two approaches to obtain desired semantics
  - The UML Meta Model

2/14

**Easy: Static Typing**

**Given:**

- $x > 0$  also well-typed for  $D_1$  (combined by  $m_1, x > 0$ )
- assignment  $!x.CI := !x.D1$  being well-typed (if  $!x.CI \vdash P$ )
- $!x.CI \vdash 0, !x.CI \vdash !x(0), !x.CI \vdash P$  being well-typed (and doing the right thing).

**Wanted:**

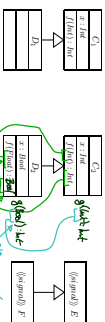
**Approach:**

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

*Handwritten notes:*  $!x.CI := !x.D_1$  is well-typed if  $!x.CI \vdash P$  and  $!x.D_1 \vdash P$ . Example:  $!x.CI := !x.D_1$  combined  $!x.D_1 \vdash P$ .  $!x.CI := !x.D_1$  combined  $!x.D_1 \vdash P$ .  $!x.CI := !x.D_1$  combined  $!x.D_1 \vdash P$ .

4/14

### Static Typing Cont'd



*Assuming  $D_1.CI \in D_2.CI$  and  $D_1.CI \in D_2.CI$  plus consistency for  $!x$ -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.

5/14

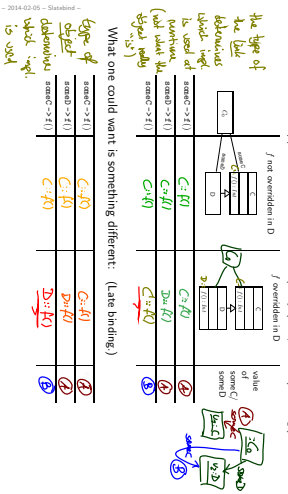
“...shall be usable...” for UML

### Excursus: Late Binding of Behavioural Features

6/14

## Late Binding

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



## Late Binding in the Standard and Programming Langs.

- 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, 2007a, 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  $C \leq D$
    - 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~~ 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;
};

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

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

int D::f(int) {
    return (3 % 2);
};
```

## Sub-Typing Principles Cont'd

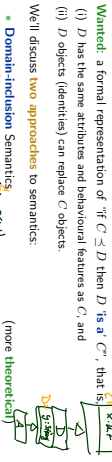
- In the standard, Section 7.3.36, "Operation":
  - Semantic Variation Points**
  - [...] When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more 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: ~~How~~ (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 Weinlein, 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 (contravariant)
  - Related: "has a weaker pre-condition," (covariant)
  - "has a stronger post-condition,"

### 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 line binding) only the state machine of the most specialised classes are running.

### Towards System States



- Wanted: a formal representation of "if  $C \leq D$  then  $D$  is a  $C$ ", that is:
  - $D$  has the same attributes and behavioural features as  $C$ , and
  - $D$  objects (identities) can replace  $C$  objects.
- We'll discuss **two approaches** to semantics:
  - Domain-Inclusion Semantics:**
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$
  - Uplink Semantics:**
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$
    - $\sigma(a) \in \sigma(b) \rightarrow \sigma(a) \in \sigma(b)$

### Domain Inclusion Structure

Let  $\mathcal{S} = (\mathcal{S}, \mathcal{E}, V, \text{attr}, \mathcal{E}, F, \text{meth}, \triangleleft)$  be a signature.

- Now a **structure**  $\mathcal{D}$
- [as before] maps types, classes, associations to domains,
  - [or **completeness**] methods to transformers,
  - [as before] identities of instances of classes not (transitively) related by generalisation are disjoint.
  - [**changed**] the identities 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**  $\sigma : \mathcal{S} \rightarrow (V \cup (\mathcal{D} \cup \mathcal{D}(R_{0,1}) \cup \mathcal{D}(R_{2,1})))$  that is, for all  $u \in \text{dom}(\sigma) \cap \mathcal{D}(C)$ ,

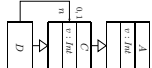
- [as before]  $\sigma(u)(v) \in \mathcal{D}(v)$  if  $v : \tau, \tau \in \mathcal{S}$  or  $\tau \in \{C_1, C_0\}$
- [**changed**]  $\text{dom}(\sigma(u)) = \bigcup_{C_0 \leq C} \text{attr}(C_0)$

**Note:** the old setting still coincides with the special case  $\triangleleft = \emptyset$

### Domain Inclusion Semantics

### Preliminaries: Expression Normalisation

- Recall:**
- we want to allow, e.g., "context  $D$  inv:  $v \triangleleft D$ ".
  - we assume **fully qualified names**, e.g.  $C::v$ .
- Intuitively,  $v$  shall denote the "most special more general"  $C::v$  according to  $\triangleleft$ .



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  $J$ ) 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 ( $\equiv$  replace by)  $C::v$ ,
- where  $C$  is the **greatest** class wrt. " $\leq$ " such that  $C \leq D$  and  $C::v \in \text{attr}(C)$ .

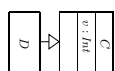
If no (unique) such class exists, the model is considered **not well-formed**: the expression is ambiguous. Then: explicitly provide the qualified name.

- Recall (part of the) OCL syntax and typing:
 
$$\begin{aligned} \text{expr} ::= & v(\text{expr}_1) && : \tau_C \rightarrow \tau(v), && \text{if } v : \tau \in \mathcal{V} \\ & | r(\text{expr}_1) && : \tau_C \rightarrow \tau_D, && \text{if } r : \tau \in \mathcal{R} \\ & | r(\text{expr}_1) && : \tau_C \rightarrow \text{Set}(\tau_D), && \text{if } r : D_{0,1} \\ & | r(\text{expr}_1) && : \tau_C \rightarrow \text{Set}(\tau_D), && \text{if } r : D_* \end{aligned}$$

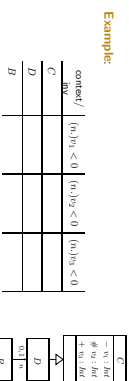
The definition of the semantics remains (textually) the same.

- We want
 
$$\text{context } D \text{ inv: } v \leq 0$$
- 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
 
$$\frac{A \vdash \text{expr}_1 : \tau_D}{A \vdash \text{expr}_1 : \tau_C}, \text{ if } C \leq D. \quad (\text{th})$$

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.



- $$\frac{A, D \vdash \text{expr}_1 : \tau_C}{A, D \vdash C::\text{val}(\text{expr}_1) : \tau}, \quad \xi = + \quad (\text{Pub})$$
- $$\frac{A, D \vdash \text{expr}_1 : \tau_C}{A, D \vdash C::\text{val}(\text{expr}_1) : \tau}, \quad \xi = \#, \tau \leq D \quad (\text{Prot})$$
- $$\frac{A, D \vdash \text{expr}_1 : \tau_C}{A, D \vdash C::\text{val}(\text{expr}_1) : \tau}, \quad \xi = -, C = D \quad (\text{Pri})$$
- $$(C::\tau : \tau, \xi, \text{obj}, P) \in \text{atr}(C).$$

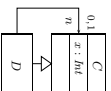


Satisfying OCL Constraints (Domain Inclusion)

- Let  $\mathcal{M} = (\mathcal{C}, \mathcal{G}, \mathcal{R}, \mathcal{H}, \mathcal{J})$  be a UML model, and  $\mathcal{S}$  a structure.
- We (continue to) say  $\mathcal{M} \models \text{expr}$  for context  $C \text{ inv: } \text{expr}_0 \in \text{Inv}(\mathcal{M})$  iff
 
$$\forall \pi = (\sigma, \varepsilon), \varepsilon \in \llbracket \mathcal{M} \rrbracket, \forall v \in \mathbb{K}, \forall u \in \text{dom}(\sigma) \cap \mathcal{G}(C):$$

$$\llbracket \text{expr} \rrbracket_{\pi}(\sigma, \{\text{self} \mapsto u\}) = 1.$$
- $\mathcal{M}$  is (still) consistent if and only if it satisfies all constraints in  $\text{Inv}(\mathcal{M})$ .

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
 
$$\sigma' = \sigma[v \mapsto \sigma(v)] \cup \llbracket \text{expr}_2 \rrbracket(\sigma)$$
- where  $u = \llbracket \text{expr}_1 \rrbracket(\sigma)$ .

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

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

$(\sigma, s) \xrightarrow{\text{trans}, \text{Send}} (\sigma', s')$  if
 

- $\exists u \in \text{Ident}(C) \cap \mathcal{D}(C) \exists \text{Msg} \in \mathcal{M}(C) : \text{msg} \in \text{recv}(s, u)$
- $u$  is stable and in state machine state  $s$ , i.e.  $\sigma(u)(\text{stable}) = 1$  and  $\sigma(u)(st) = s$ ,
- a transition is enabled, i.e.
 
$$\exists (s', F, \text{expr}, \text{act}, s'') \in \text{SM}(C) : F = E \wedge \text{Trans}[s] = 1$$
 where  $\delta = \sigma(u, \text{param}_s \mapsto \text{val})$ .

and

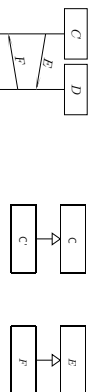
- $(\sigma', s')$  results from applying  $\text{act}$  to  $(\sigma, s)$  and removing  $\text{msg}$  from the stack, i.e.
 
$$(\sigma', s') = \text{act}(\delta, \sigma \in \text{val})$$

where  $k$  depends:

- If  $u$  becomes stable in  $s'$ , then  $\delta = 1$ . It does become stable if and only if there is no transition without trigger enabled for  $u$  in  $(\sigma', s')$ .
- Otherwise  $\delta = 0$ .
- Consumption of  $\text{msg}$  and the side effects of the action are observed, i.e.
 
$$\text{cons} = \{(u, (E, \sigma(u))), \text{Send} = \text{Obs}_{s, s'}(E, \sigma(u))\}$$

25/74

### 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.
 
$$\sigma, \text{cons}, \text{Send} \models_{\beta} E_{x, \theta}$$

if and only if

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

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

26/74

### Uplink Semantics

27/74

### 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) preprocessing to make appropriate use of that association, e.g. rewrite  $(C++)$

in  $D$  to  $x = 0;$   
 $\text{uplink}_C \rightarrow x = 0;$

28/74

### Pre-Processing for the Uplink Semantics

- For each pair  $C \triangleleft D$ , extend  $D$  by a (fresh) association  $\text{uplink}_C : C$  with  $\mu = [1, 1]$ ;  $\xi = +$  (Exercise: public necessary?)
- Given expression  $v$  (or  $j$ ) in the context of class  $D$ ,
  - let  $C$  be the **smallest** class wrt. " $\preceq$ " such that
  - $C \preceq D$ , and
  - $C::x \in \text{attr}(D)$
- then there exists (by definition)  $C \triangleleft C_1 \triangleleft \dots \triangleleft C_n \triangleleft D$ ,
  - **normalise**  $v$  to  $(=$  replace by)  $\text{uplink}_{C_n} \rightarrow \dots \rightarrow \text{uplink}_{C_1} \cdot C::x$

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

29/74

### 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

30/74

- Let  $\mathcal{M} = (\mathcal{G}, \mathcal{D}, \mathcal{R}, \mathcal{I}, \mathcal{J})$  be a UML model, and  $\mathcal{G}$  a structure.
- We (continue to) say  $\mathcal{M} \models_{\text{expr}}$  for
- context  $C$ ,  $\text{inv} : \text{expr}_0 \in \text{Inv}(\mathcal{M})$
- if and only if
  - $\forall \tau = \langle \alpha \rangle, \alpha \in \llbracket \mathcal{M} \rrbracket$
  - $\forall i \in \mathbb{N}$
  - $\forall u \in \text{dom}(\alpha_i) \cap \mathcal{D}(C)$  :
  - $\llbracket \text{expr} \rrbracket[\alpha_i, [\text{self} \mapsto u]] = 1$ .
- $\mathcal{M}$  is (still) consistent if and only if it satisfies all constraints in  $\text{Inv}(\mathcal{M})$ .

- What has to change is the create transformer:
  - $\text{create}_C(C, \text{expr}, v)$
- Assume,  $C$ 's inheritance relations are as follows.
  - $C_{1,1} \triangleleft \dots \triangleleft C_{1,m} \triangleleft C$
  - $\dots$
  - $C_{n,1} \triangleleft \dots \triangleleft C_{n,m} \triangleleft C$
- Then, we have to
  - create one fresh object for each part, e.g.
    - $\forall i_1, 1 \dots m, \text{inv}_1, \dots, \text{inv}_m, 1 \dots m, n, m$
  - set up the uplinks recursively, e.g.
    - $\sigma(\text{inv}_1, 2)(\text{uplink}_{C_{1,1}}) = \text{inv}_1, 1$
- And, if we had constructors, be careful with their order.

- Employ something similar to the "worsepec" trick (in a minute!). But the result is typically far from concise. (Related to OCL's `!skindof()` function, and RTTI in C++)

Domain Inclusion vs. Uplink Semantics

```

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

Casts in Transformers

```

• C c:
• D d:
• Identity upcast (C++):
    // assign address of 'd' to pointer 'cp'
    C* cp = &d;
• Identity downcast (C++):
    // assign address of 'd' to pointer 'dp'
    D* dp = (D*)cp;
• Value upcast (C++):
    // copy attribute values of 'd' into 'c'; or
    // move precise, the values of the C-part of 'd'
    c = *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 $\mathcal{D}(D) \subset \mathcal{D}(C)$	easy: By pre-processing $C^* cp = \text{dup}(\&d, C)$
$D^* dp = (D^*)cp;$	easy: the value of $cp$ is in $\mathcal{D}(D) \cap \mathcal{D}(C)$ because the pointed-to object is a $D$ . Otherwise, error condition. (See next slide.)	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 $C \subseteq D$ ) $(C) \langle \cdot, \cdot \rangle : \mathcal{D} \times \Sigma \rightarrow \Sigma_{\text{inv}(C)}$ $(u, v) \mapsto \sigma(u)_{\text{inv}(C)}$ . Note: $\sigma^* = \sigma_{\text{inv}(C)} \circ \sigma^*(u)$ is not type-compatible!	easy: By pre-processing $c = \#(\text{dup}(\&d, C))$

### Identity Downcast with Uplink Semantics

- **Recall** (C++):  $D \leq C$ :  $C^* \text{ cp} = \text{kd}$ ;  $D^* \text{ dp} = (D) \text{ cp}$ :
- **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
 
$$\text{all} \in \mathcal{D}, \quad \mathcal{D}(\text{all}) = \bigcup_{C \in \mathcal{C}} \mathcal{D}(C)$$
- In each  $\leq$ -minimal class have associations "astepc" pointing to **most specialised** slices, plus information of which type that slice is.
- Then **downcast** means, depending on the **rootstep** type (only finitely many possibilities), **going down and then up** as necessary, e.g.
 

```

class C {
    astepc:rootstepc,type {}
}
dp = cp -> mostspec -> uplink0 -> ... -> uplinkn -> uplinkn -> uplinkn
    
```

### Meta-Modelling: Idea and Example

### 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 need 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 **doesn't** improve conciseness **so dramatically**.
- As long as we're "early binding", that is...

### Meta-Modelling: Why and What

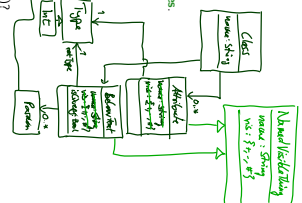
- **Meta Modelling** is one major prerequisite for understanding the standard documents [OMG, 2007a, OMG, 2007b] and the MDA ideas of the OMG.
  - The idea is **simple**:
  - if a **modelling language** is about modelling things,
  - and if UML models are and comprise things,
  - then why not **model** 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.

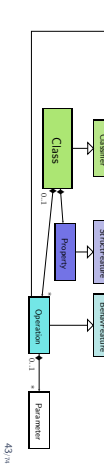
### Domain Inclusion vs. Uplink Semantics: Motives

- **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: 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)?





43/n

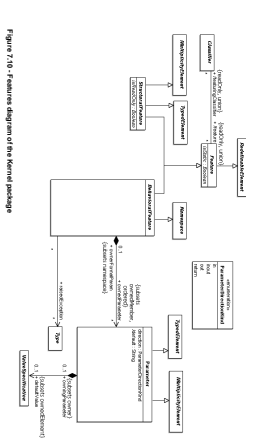
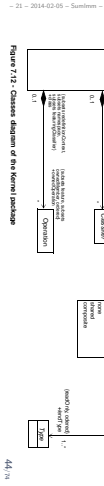


Figure 7.10 - Features diagram of the kernel package



44/n

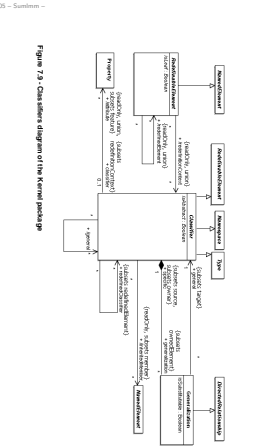


Figure 7.9 - Classifiers diagram of the kernel package



Figure 7.11 - Namespaces diagram of the kernel package

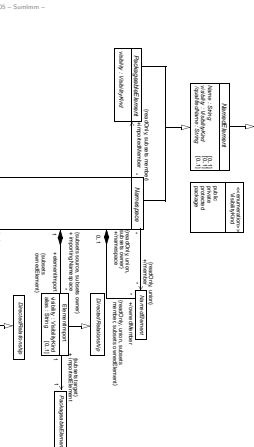


Figure 7.4 - Namespaces diagram of the kernel package



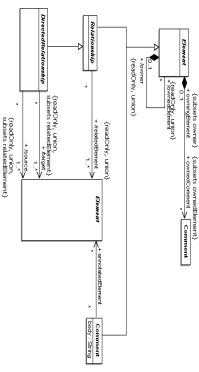


Figure 7.2 - Root Diagram of the Kernel package

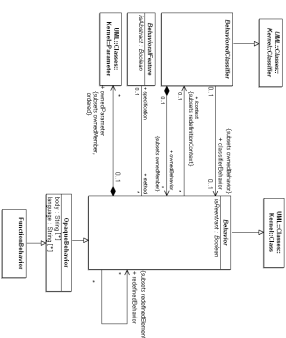


Figure 11.8 - Common Kernel

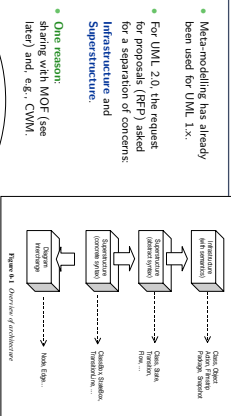


Figure B1 - Overview of Architecture

- Meta-modeling 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.

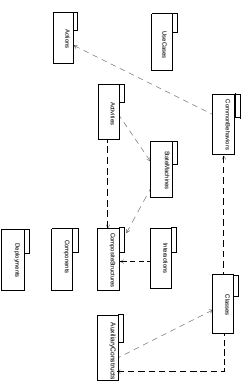
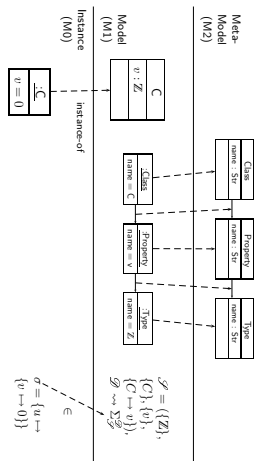
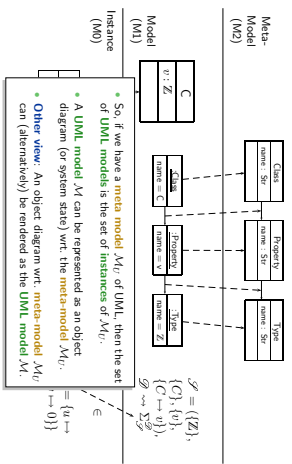


Figure 7.5 - The top-level package structure of the UML 2.1.3 Superstructure

Meta-Modeling: Principle





- 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,
  - 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() -> included self" [OMG, 2007b, 53]
- The other way round:
  - Given a **UML model**  $M$ , unfold it into an object diagram  $O$  wrt.  $M$ .
  - If  $O$  is a **valid** object diagram of  $M$  (i.e. satisfies all invariants from  $Inv(M)$ ), 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	2
3. Language Notation	2
4. Notations and Formal Conventions	2
5. Normative References	6
6. Terms and Definitions	10
7. Symbols	10
8. Additional Information	10
9. Architectural Alignment	10
10. Other Information	11
11. The UML Metamodel	11
12. The UML Modeling Language	12
13. The UML Diagramming Language	13
14. The UML Diagramming Language	14
15. The UML Diagramming Language	15
16. The UML Diagramming Language	16
17. The UML Diagramming Language	17
18. The UML Diagramming Language	18
19. The UML Diagramming Language	19
20. The UML Diagramming Language	20
21. The UML Diagramming Language	21
Part I - Structure	21
7. Classes	21

Table of Contents	
1. Scope	1
2. Conformance	2
3. Language Notation	2
4. Notations and Formal Conventions	2
5. Normative References	6
6. Terms and Definitions	10
7. Symbols	10
8. Additional Information	10
9. Architectural Alignment	10
10. Other Information	11
11. The UML Metamodel	11
12. The UML Modeling Language	12
13. The UML Diagramming Language	13
14. The UML Diagramming Language	14
15. The UML Diagramming Language	15
16. The UML Diagramming Language	16
17. The UML Diagramming Language	17
18. The UML Diagramming Language	18
19. The UML Diagramming Language	19
20. The UML Diagramming Language	20
21. The UML Diagramming Language	21
Part I - Structure	21
7. Classes	21

Table of Contents	
1. Scope	1
2. Conformance	2
3. Language Notation	2
4. Notations and Formal Conventions	2
5. Normative References	6
6. Terms and Definitions	10
7. Symbols	10
8. Additional Information	10
9. Architectural Alignment	10
10. Other Information	11
11. The UML Metamodel	11
12. The UML Modeling Language	12
13. The UML Diagramming Language	13
14. The UML Diagramming Language	14
15. The UML Diagramming Language	15
16. The UML Diagramming Language	16
17. The UML Diagramming Language	17
18. The UML Diagramming Language	18
19. The UML Diagramming Language	19
20. The UML Diagramming Language	20
21. The UML Diagramming Language	21
Part I - Structure	21
7. Classes	21
8. Components	143
9. Composite Structures	141
10. Deployment	181

Table of Contents	
1. Scope	1
2. Conformance	2
3. Language Notation	2
4. Notations and Formal Conventions	2
5. Normative References	6
6. Terms and Definitions	10
7. Symbols	10
8. Additional Information	10
9. Architectural Alignment	10
10. Other Information	11
11. The UML Metamodel	11
12. The UML Modeling Language	12
13. The UML Diagramming Language	13
14. The UML Diagramming Language	14
15. The UML Diagramming Language	15
16. The UML Diagramming Language	16
17. The UML Diagramming Language	17
18. The UML Diagramming Language	18
19. The UML Diagramming Language	19
20. The UML Diagramming Language	20
21. The UML Diagramming Language	21
Part I - Structure	21
7. Classes	21
8. Components	143
9. Composite Structures	141
10. Deployment	181

Reading the Standard Cont'd

Table with 2 columns: Standard Number and Standard Description. Includes sections for 12A, 12B, 12C, 12D, 12E, 12F, 12G, 12H, 12I, 12J, 12K, 12L, 12M, 12N, 12O, 12P, 12Q, 12R, 12S, 12T, 12U, 12V, 12W, 12X, 12Y, 12Z.

Reading the Standard Cont'd

Table with 2 columns: Standard Number and Standard Description. Includes sections for 12A, 12B, 12C, 12D, 12E, 12F, 12G, 12H, 12I, 12J, 12K, 12L, 12M, 12N, 12O, 12P, 12Q, 12R, 12S, 12T, 12U, 12V, 12W, 12X, 12Y, 12Z.

Reading the Standard Cont'd

Table with 2 columns: Standard Number and Standard Description. Includes sections for 12A, 12B, 12C, 12D, 12E, 12F, 12G, 12H, 12I, 12J, 12K, 12L, 12M, 12N, 12O, 12P, 12Q, 12R, 12S, 12T, 12U, 12V, 12W, 12X, 12Y, 12Z.

Table with 2 columns: Standard Number and Standard Description. Includes sections for 12A, 12B, 12C, 12D, 12E, 12F, 12G, 12H, 12I, 12J, 12K, 12L, 12M, 12N, 12O, 12P, 12Q, 12R, 12S, 12T, 12U, 12V, 12W, 12X, 12Y, 12Z.

Table with 2 columns: Standard Number and Standard Description. Includes sections for 12A, 12B, 12C, 12D, 12E, 12F, 12G, 12H, 12I, 12J, 12K, 12L, 12M, 12N, 12O, 12P, 12Q, 12R, 12S, 12T, 12U, 12V, 12W, 12X, 12Y, 12Z.

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 ...

59/74

## 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. [Buscherhölle and Oelertik, 2008]

60/74

## 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

62/74

## Benefits for Modelling Tools

- The meta-model *M<sub>0</sub>* of UML **immediately** provides a **data-structure** representation for the abstract syntax (~ 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 such 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.

63/74

63/74

## 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. — XML Metadata Interchange (XMI)
- **Note**: A priori, there is no graphical information in XMI (it is only abstract syntax like our signatures) —> OMG Diagram Interchange.
- **Note**: There are slight ambiguities in the XML 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 XML compatibility doesn't necessarily refer to Diagram Interchange.
- **To reiterate**: this is **generic** for all MOF-based modelling languages such as UML, QVT, etc. And also for **Domain Specific Languages** which don't even exist yet.

64/74

64/74

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

- 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**.

- 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. [Scah and Valter, 2005])

- 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.

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

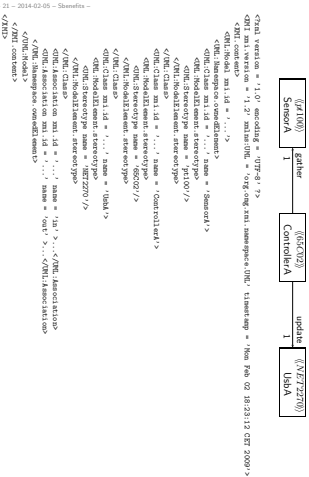
- There are manifold applications for model-to-model transformations:
  - For instance, tool support for **re-factoring**s, like moving common attributes upwards the inheritance hierarchy.This can now be defined as **graph-rewriting** rules on the level of MOF.  
The graph to be rewritten is the UML 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**.

71/n

## Example: Model and XML



72/n

## References

73/n

## References

[Bachemirble and Oelertik, 2008] Bachemirble, R. and Oelertik, J. (2008). Rich meta object facility. In Proc. Int. Workshop UML and Formal Methods

[Fraser and Wehrman, 2000] Fraser, C. and Wehrman, H. (2000). Behavioural labelling relations for object-oriented systems. In Kus, T., editor. *ADSLT*, number 1818 in Lecture Notes in Computer Science. Springer, Berlin.

[OMG, 2008] OMG. (2008). Uml 2.0 proposal of the 2U group, version 0.2. <http://www.zenworks.org/uml2anastasis>.

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