

Software Design, Modelling and Analysis in UML

Lecture 20: Inheritance I

2014-02-03

Prof. Dr. Andreas Podolski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

- **Last Lecture:**
 - Live Sequence Charts Semantics

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions:
 - What's the Lifesov Substitution Principle?
 - What is lifecycle binding?
 - What is the subject, what the uplink semantics of inheritance?
 - What's the effect of inheritance on LSCs, State Machines, System States?
 - What's the idea of Meta-Modeling?

- **Content:**
 - Quickly: Behavioural Features, Active vs. Passive
 - Inheritance in UML: concrete syntax
 - Lifesov Substitution Principle — desired semantics
 - Two approaches to obtain desired semantics
 - The UML Meta Model

2/99

Active and Passive Objects [Harel and Gery, 1997]

What about non-Active Objects?

Recall:

- We're **still** working under the assumption that all classes in the class diagram (and thus all objects) are **active**.
- That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

But the world doesn't consist of **only active objects**.

For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control.

So we have to address questions like:

- Can we send events to a non-active object?
- And if so, when are these events processed?
- etc.

4/99

Active and Passive Objects: Nomenclature

[Harel and Gery, 1997] propose the following (orthogonal) notions:

- A class (and thus the instances of this class) is either **active** or **passive** as declared in the class diagram.
- An **active** object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
- A **passive** object doesn't.
- A class is either **reactive** or **non-reactive**.
- A **reactive** class has a (non-trivial) state machine.
- A **non-reactive** one hasn't.

Which combinations do we understand?

	active	passive
reactive	✓	?
non-reactive	(?)	(?)

5/99

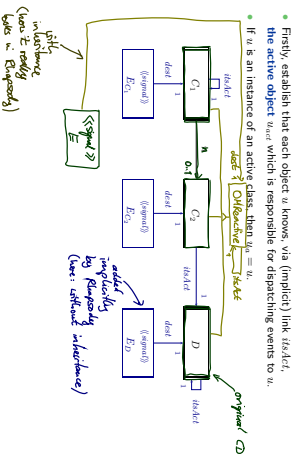
Passive and Reactive

- So why don't we understand passive/reactive?
- Assume passive objects u_1 and u_2 , and active object u_3 , and that there are events in the ether for all three.
- Which of them (can) start a run-to-completion step...? Do run-to-completion steps still interleave...?

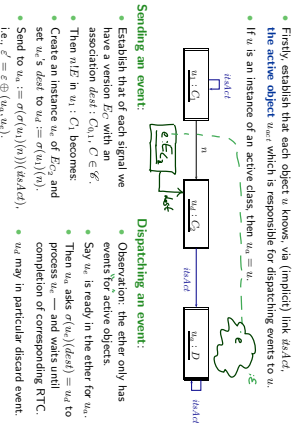
Reasonable Approaches:

- **Avoid** — for instance, by
 - require that **reactive implies active** for model well-formedness.
 - requiring for model well-formedness that events are **never sent** to instances of non-reactive classes.
- **Explain** — here, (following [Harel and Gery, 1997])
 - Delegate all dispatching of events to the active objects.

6/99



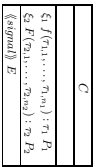
7/9



7/9

And What About Methods?

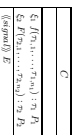
- In the current setting, the (local) state of objects is only modified by actions of transitions, which we abstract to transformers.
- In general, there are also **methods**.
- UML follows an approach to separate
 - the **interface declaration** from
 - the **implementation**.
- In C++inger, distinguish **declaration** and **definition** of method.
- In UML, the former is called **behavioural feature** and can (roughly) be
 - a **call interface** $f(\tau_1, \dots, \tau_n) : \tau$
 - a **signal name** E



9/9

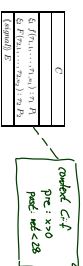
Behavioural Features

- Semantics:**
- The **implementation** of a behavioural feature can be provided by:
 - An **operation**.
- In our setting, we simply assume a transformer like T_f .
- It is then, e.g., clear how to admit method calls as actions on transitions, function composition of transformers (Clear but tedious: non-termination).
- In a setting with Java as action language: operation is a method body $\mu \rightarrow \mathcal{E}$.
- The class **state-machine** ("triggered operation")
- calling F with α parameters for a stable instance of C creates an auxiliary event F and dispatches it (bypassing the ether);
 - Transition actions may fill in the return value.
 - On completion of the RTC step, the call returns.
 - For a non-stable instance, the caller blocks until stability is reached again.



10/9

And What About Methods?



- Behavioural Features: Visibility and Properties**
- Visibility:**
 - Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.
 - Useful properties:**
 - concurrency** — is thread safe
 - concurrent** — some mechanism ensures/should ensure mutual exclusion
 - guarded** — is not thread safe, users have to ensure mutual exclusion
 - isQuery** — doesn't modify the state space (thus thread safe)
 - For simplicity, we leave the notion of steps unbounded, we construct our semantics around state machines. Yet we could explain pre/post in OCL (if we wanted to).

11/9

Semantic Variation Points

Pessimistic view: They are legion...

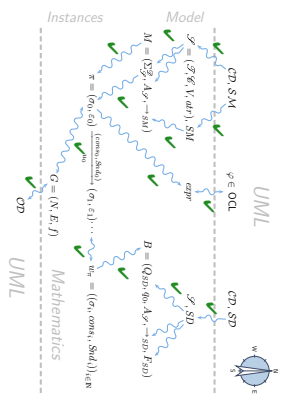
- For instance,
 - allow absence of initial pseudo-states
 - can then “be” in enclosing state without being in any substate, or assume one of the children states non-deterministically
 - (implicitly) enforce determinism, e.g. by considering the order in which things have been added to the CASE tool’s repository, or graphical order
- allow true concurrency

Exercise: Search the standard for “semantical variation point”.

- (Crane and Dingel, 2007), e.g. provide an in-depth comparison of Statemate, UML, and Rhapsody state machines — the bottom line is:
 - the intersection is not empty
 - (i.e. there are pictures that mean the same thing to all three communities) (i.e. for each pair of communities exists pictures meaning different things)
 - none is the subset of another

Optimistic view: tools exist with complete and consistent code generation.

Course Map



Inheritance: Syntax

ABSTRACT SYNTAX

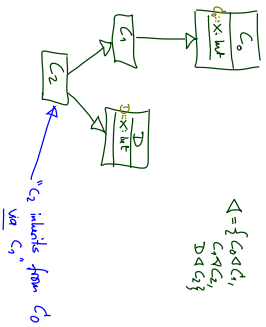
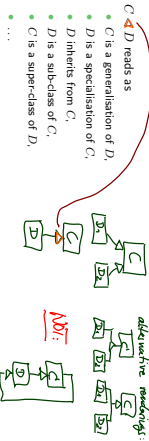
Recall: a signature (with signals) is a tuple $\mathcal{S} = (\mathcal{S} \subseteq V, \text{attr}, \delta, F, \text{mb}, \triangleleft)$.

Now (finally), extend to $\mathcal{S} = (\mathcal{S} \subseteq V, \text{attr}, \delta, F, \text{mb}, \triangleleft)$

where F/mb are methods, analogously to attributes and

$$\triangleleft \subseteq (\mathcal{S} \times \mathcal{S}) \cup (C \times C) \cup (C \times \mathcal{S})$$

is a generalisation relation such that $C \triangleleft^+ C'$ for no $C \in \mathcal{C}$ (“acyclic”)



Recall: Reflexive, Transitive Closure of Generalisation

Definition. Given classes $C_0, C_1, D \in \mathcal{C}$, we say D inherits from C_0 via C_1 if and only if there are $C_0^1, \dots, C_0^n, C_1^1, \dots, C_1^m \in \mathcal{C}$ such that

$$C_0 \triangleleft C_0^1 \triangleleft \dots \triangleleft C_0^n \triangleleft C_1 \triangleleft C_1^1 \triangleleft \dots \triangleleft C_1^m \triangleleft D.$$

We use ' \triangleleft^* ' to denote the reflexive, transitive closure of ' \triangleleft '.

- In the following, we assume
 - that all attribute (method) names are of the form

$$C::n, C \in \mathcal{C} \cup \delta \quad (C::f, C \in \mathcal{C}),$$
 - that we have $C::n \in \text{attr}(C)$, resp. $C::f \in \text{meth}(C)$ if and only if ' n ' (f) appears in an attribute (method) compartment of C in a class diagram.

We still want to accept "context C inv: $v \triangleleft 0$ ", which v is meant? Later!

Inheritance: Desired Semantics

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from subtypes, which in the OO domain is closely related to inheritance: The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994]. (Liskov Substitution Principle (LSP))

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

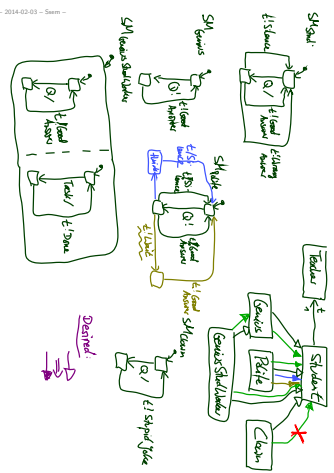
$$S \text{ sub-type of } T \iff \forall o_1 \in S \exists o_2 \in T \forall P. o_1 \bullet [P] \leq o_2 \bullet [P]$$

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one expects from subtypes, which in the OO domain is closely related to inheritance: The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994]. (Liskov Substitution Principle (LSP))

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

In other words: [Fischer and Weinheim, 2000]
 "An instance of the sub-type shall be usable whenever an instance without a client being able to tell the difference."
 So, what's "usable"? Who's a "client"? And what's a "difference"?



"...shall be usable..."?



OCL: $\text{context } C \text{ inv: } x \geq 0$

Sequence Diagrams

Actions:

- $\text{hs}(C.x) = 0$ *unusable*
- $\text{hs}(C.f())$ *usable*
- $\text{hs}(C1.f)$ *depend*

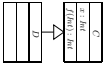
Triggers:

- $E[...]$...

“...a client...”?

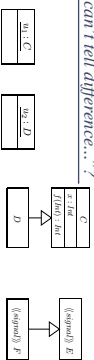
“An instance of the sub-type shall be usable whenever an instance of the super-type was expected, without a client being able to tell the difference.”

- **Narrow interpretation:** another object in the model
- **Wide interpretation:** another modeller.



22/99

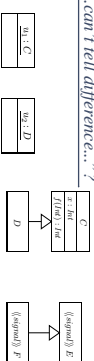
“...can't tell difference...”?



- **OCL:** $\llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_1, \theta)$ vs $\llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_2, \theta)$

23/99

“...can't tell difference...”?



- **Triggers, Actions:** if

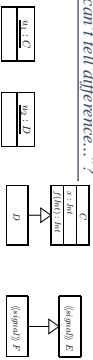
is possible then

$$\frac{\llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_1, \theta) \quad \llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_2, \theta)}{\llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_1, \theta) \quad \llbracket \text{Context } C' \text{ inv: } x > 0 \rrbracket (c_2, \theta)}$$

should be possible — sub-type does less on inputs of super-type
for some $v_1 \in \text{EX}(C)$ and a proper definition of $\llbracket \cdot \rrbracket$

24/99

“...can't tell difference...”?



- **Sequence Diagram:** $w \in \text{EX}(C)$ implies $w \in \text{EX}(D)$.

25/99

Motivations for Generalisation

- Re-use.
- Sharing.
- Avoiding Redundancy.
- Modularisation.
- Separation of Concerns.
- Abstraction.
- Extensibility.
- ...

— See [repehde](#) on object-oriented analysis, development, programming.

26/99

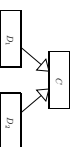
What Does [Fischer and Wehrheim, 2000] Mean for UML?

“An instance of the sub-type shall be usable whenever an instance of the super-type was expected, without a client being able to tell the difference.”

- Wanted: sub-typing for UML.
- With

we don't even have usability.

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



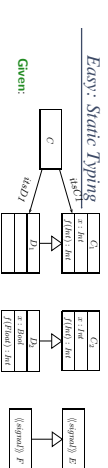
would ensure D_1 is a sub-type of C :

- that D_1 objects can be used interchangeably by everyone who is using C 's,
- is not able to tell the difference (i.e. see unexpected behaviour).

27/99

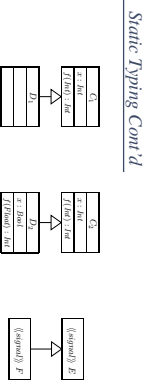
“...shall be usable...” for UML

Excursus: Late Binding of Behavioural Features



- Given:**
- $x > 0$ also well-typed for D_1
 - assignment $(\lambda x.C1, f(0), \lambda x.C1 \vdash F$ being well-typed (and doing the right thing).
- Wanted:**
- assignment $(\lambda x.C1, f(0), \lambda x.D1 \vdash F$ being well-typed

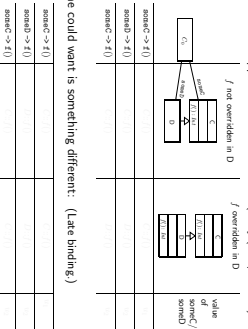
- Approach:**
- Simply define it as being well-typed
 - adjust system state definition to do the right thing.



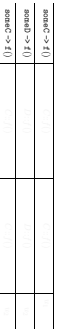
- Notions (from category theory):**
- invariance,
 - covariance,
 - contravariance.
- We could call, e.g. a method, **sub-type preserving**, if and only if it
- accepts **more general** types as input (contravariant),
 - provides a **more specialised** type as output (covariant).
- This is a notion used by many programming languages — and easily type-checked.

Late Binding

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



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



Late Binding in the Standard and Programming Lang.

- In the standard, Section 11.3.10, “CallOperatorAction”:
 - “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?

- 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 indicate the effect of "early binding"

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

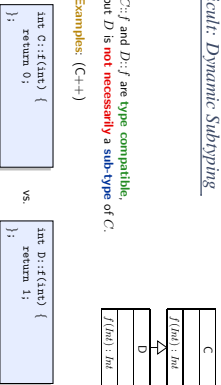
Note: late binding typically applies only to **methods not to attributes**.
 (But: getter/setter methods have been invented recently)

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

Difficult: Dynamic Subtyping

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

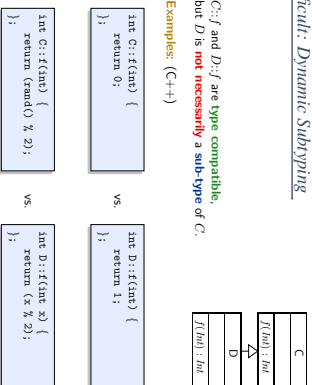
• Examples: (C++)



Difficult: Dynamic Subtyping

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

• Examples: (C++)



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.

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]

- 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 (i) is no longer a matter of simple type-checking!

- 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!

- 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.

- 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!

- 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," (covariant)
"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.



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).

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.

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

Now a **structure** \mathcal{D}

- [as before] maps types, classes, associations to domains,
- [for 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.

$$\forall C \in \mathcal{C} : \mathcal{D}(C) \supseteq \bigcup_{C' \triangleleft C} \mathcal{D}(C')$$

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

Domain Inclusion System States

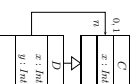
Now: a **system state** of \mathcal{S} wrt. \mathcal{D} is a **type-consistent mapping**

$$\sigma : \mathcal{D}(\mathcal{C}) \mapsto (V \mapsto (\mathcal{D}(\mathcal{S}) \cup \mathcal{D}(R_{0,1}) \cup \mathcal{D}(R_{2,1})))$$

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

- [as before] $\sigma(u)(v) \in \mathcal{D}(r)$ if $v : \tau, \tau \in \mathcal{S}$ or $\tau \in \{C_+, C_{0,1}\}$
- [changed] $\text{dom}(\sigma(u)) = \bigcup_{C_0 \triangleleft C} \text{attr}(C_0)$

Example:



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

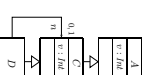
Preliminaries: Expression Normalisation

Recall:

- we want to allow, e.g., “context D inv : $t \triangleleft 0'$ ”.
- we assume fully qualified names, e.g. $C::x$.

Intuitively, w shall denote the

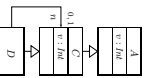
“most special more general” $C::w$ according to \triangleleft .



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 \triangleleft .



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

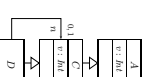
- Given expression v (or J) in context of class D , as determined by, e.g.
 - by the (type of the) navigation expression prefix, or
 - by the class, the state-machine where the action occurs belongs to,
 - similar for method bodies,
- normalise** v to (\equiv replace by) $C::v$,
- where C is the **greatest**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

43/99

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 \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. " \triangleleft " such that
 - $C \preceq D$ 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**.

43/99

More Interesting: Well-Typed-ness

- We want

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

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

- So, add a (first) new typing rule

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

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.

45/99

Well-Typed-ness with Visibility Cont'd

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

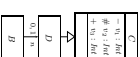
$$\frac{A, D \vdash \text{expr} : \tau_C}{A, D \vdash C::v(\text{expr}) : \tau^*} \quad \xi = \#; \tau \leq \tau^* \quad (\text{Prot})$$

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

($C::v : \tau; \xi; v_0, P \in \text{attr}(C)$.)

Example:

context/	(a)v < 0	(a)v_0 < 0	(a)v_0 < 0
inv			
C			
D			



46/99

OCL Syntax and Typing

- Recall (part of the) OCL syntax and typing

$$\text{expr} ::= v(\text{expr}_1) : \tau_C \rightarrow \tau^*(v), \quad \text{if } v : \tau \in \mathcal{D}$$

$$| \tau(\text{expr}_1) : \tau_C \rightarrow \tau_D, \quad \text{if } \tau : D_{0,1}$$

$$| \tau(\text{expr}_1) : \tau_C \rightarrow \text{Self}(\tau_D), \quad \text{if } \tau : D_1$$

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

44/99

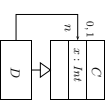
Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathcal{S}, \mathcal{D}, \mathcal{R}, \mathcal{J})$ be a UML model, and \mathcal{D} a structure.
- We (continue to) say $\mathcal{M} \models \text{expr}$ for context C inv: $\text{expr} \in \text{Inv}(\mathcal{M})$ iff

$$\forall \pi = (\sigma, \varepsilon)_{i \in R} \in \llbracket \mathcal{M} \rrbracket \quad \forall i \in \mathbb{N} \quad \forall u \in \text{dom}(\sigma) \cap \mathcal{D}(C) : \\ \mathbb{I}[\text{expr}]_i(\sigma, \{\text{self} \mapsto u\}) = 1$$

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

Example:



47/99

Transformers (Domain Inclusion)

- Transformers also remain the same, e.g. [VL 12, p. 18]
- $update(cexpr_1, u, cexpr_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon)$
- with

$$\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto \llbracket cexpr_2 \rrbracket(\sigma)]]$$
- where $u = \llbracket cexpr_1 \rrbracket(\sigma)$.

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

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

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

- $\exists h \in \text{dom}(\sigma) \cap \mathcal{G}(C) \exists \text{sig} \in \mathcal{G}(\sigma) : \text{sig} \in \text{mod}(c, u)$
- u is stable and its state machine state s , i.e. $\sigma(u)(\text{state}(s)) = 1$ and $\sigma(u)(\text{id}) = s$.
- a transition is enabled, i.e.

$$\exists (s, F, \text{exp}, \text{act}, \delta) \in \text{SM}(s) : F = E \wedge \llbracket \text{exp} \rrbracket(\sigma) = 1$$

where $\sigma = \sigma[u, \text{perm}_s \mapsto u]$.

and

- (σ', ε') results from applying $\tau_{c, u}$ to (σ, ε) and removing u, δ from the other, i.e.

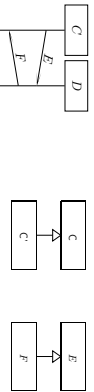
$$(\sigma', \varepsilon') = \text{fst}(\tau_{c, u}(\sigma, \varepsilon))$$

where δ depends:

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

$$\text{cons} = ((u, \delta, \sigma(u)), \text{Stim} = \text{Obs}_{\text{stim}}(\delta, \varepsilon \oplus \text{sig}))$$

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.

$$\sigma, \text{cons}, \text{Stim} \models_{\beta} F_{x, y}$$
- 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.

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 $(C++)$

$$x = 0;$$
- in D to

$$\text{upLink}_C \rightarrow x = 0;$$

- For each pair $C \triangleleft D$, extend D by a (fresh) association $uplink_C : C$ 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 " \leq " such that
 - $C \leq D$, 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) $uplink_{C_n} \rightarrow \dots \rightarrow uplink_{C_1} C::x$
- Again: if no (unique) smallest class exists, the model is considered **not well-formed**: the expression is ambiguous.

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

- Let $\mathcal{M} = (\mathcal{O}, \mathcal{O}, \mathcal{S}, \mathcal{K}, \mathcal{J})$ be a UML model, and \mathcal{Q} 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

$$\forall \pi = (\alpha)_k, \alpha \in \llbracket \mathcal{M} \rrbracket$$

$$\forall i \in \mathbb{N}$$

$$\forall u \in \text{dom}(\alpha) \cap \mathcal{O}(C) :$$

$$\llbracket \text{expr}_0 \rrbracket(\alpha, \text{self} \mapsto u) = 1.$$
- \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: $\text{create}(C, \text{expr}, v)$
- Assume, C 's inheritance relations are as follows:

$$C_{n+1} \triangleleft \dots \triangleleft C_{i+1} \triangleleft C_i$$

$$\dots$$

$$C_{n+1} \triangleleft \dots \triangleleft C_{m+1} \triangleleft C$$
- Then, we have to
 - create one fresh object for each part, e.g. $u_{1,1}, \dots, u_{1,n}, \dots, u_{m,1}, \dots, u_{m,m}$
 - 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.

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

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

	Domain Inclusion	Uplink
C \rightarrow cp = &c;d	easy: Immediately compatible (in underlying system state) because &c;d yields an identity from $\mathcal{D}(D) \subset \mathcal{D}(C)$.	easy: By pre-processing, C \rightarrow cp = d.uplink _c ;
D \rightarrow 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.	difficult: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)
c = d;	hit difficult: set (for all C \leq D) $(C) \setminus \{ \cdot \} : \tau_D \times \Sigma \rightarrow \Sigma_{\text{link}(C)}$ $(\tau_c, \sigma) \mapsto \sigma(\alpha)_{\text{link}(C)}$. Note: $\sigma' = \sigma(\alpha)_{\text{link}(C)}$ is not type-compatible!	easy: By pre-processing, c = *(d.uplink _c);

- **Recall** (C \rightarrow D): D: d; C \rightarrow cp = &c;d; D \rightarrow 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
 - $\text{all} \in \mathcal{D}'$, $\mathcal{D}'(\text{all}) = \bigcup_{C \leq D} \mathcal{D}(C)$
- In each Σ -minimal class have associations "base_{spec}" pointing to most specialised slices, plus information on which type that slice is;
- Then downcast means, depending on the base_{spec} type (only finitely many possibilities), going down and then up as necessary, e.g.


```

ast::downcast<T>() {
  case C:
    dp = cp -> basespec -> uplinkD_1 -> ... -> uplinkD_n -> uplinkD_n;
  ...
}
            
```

Domain Inclusion vs. Uplink Semantics: Differences

- **Note:** The uplink semantics views inheritance as an abbreviation;
 - We only need to touch semantics (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 also **doesn't** improve conciseness **too** dramatically;
- As long as we're "early binding", that is...

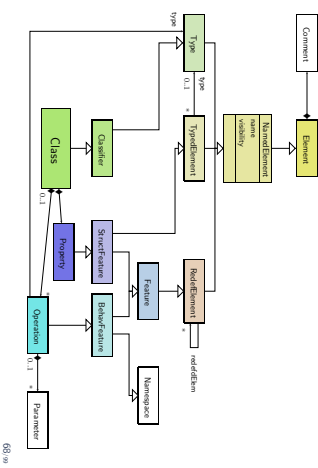
Domain Inclusion vs. Uplink Semantics: Motives

- **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-Modeling** 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?

UML Meta-Model: Extract



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

Classes [OMG, 2007b, 32]

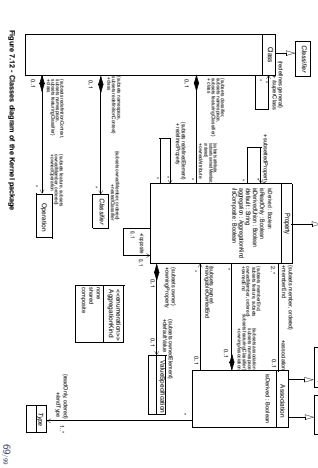


Figure 7-12: Classes diagram of the kernel package

- For example, let's consider a class.
- A **class** has (on a superficial level)
 - a **name**,
 - any number of **attributes**,
 - any number of **behavioral features**.
- Each of the latter two has
 - a **name** and
 - a **visibility**.
- Behavioral features in addition have
 - a boolean attribute **isQuery**,
 - any number of **parameters**,
 - a **return type**.
- Can we model this (in UML, for a start)?

Operations [OMG, 2007b, 31]

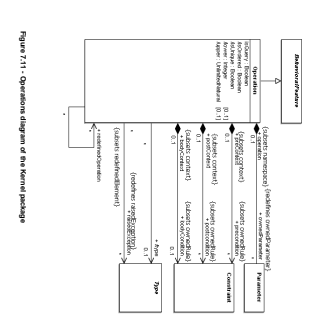


Figure 7-11: Operations diagram of the kernel package

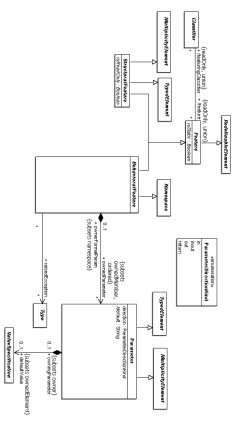


Figure 7.10 - Root diagram of the Kernel package

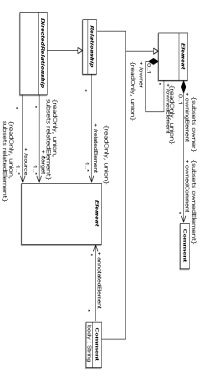


Figure 7.21 - Root diagram of the Kernel package

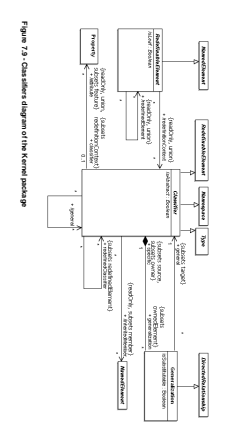


Figure 7.3 - Classifier diagram of the Kernel package

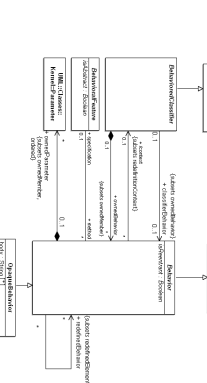


Figure 11.8 - Common Blocker

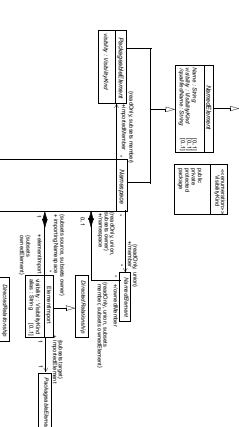


Figure 7.4 - Namespace diagram of the Kernel package

- 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.

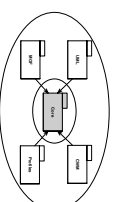
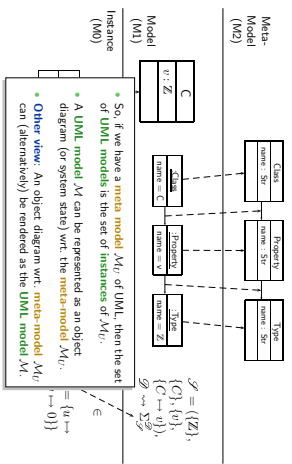


Figure 8.1 - Overview of Infrastructure



- 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_M wrt. M_{UML}. If O_M is a valid object diagram of M_{UML} (i.e. satisfies all invariants from Inv(M_{UML})), then M is a well-formed UML model.
- That is, if we have an object diagram validity checker for the meta-modelling language, then we have a well-formedness checker for UML models.

- 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_M wrt. M_{UML}. If O_M is a valid object diagram of M_{UML} (i.e. satisfies all invariants from Inv(M_{UML})), then M is a well-formed UML 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_M wrt. M_{UML}. If O_M is a valid object diagram of M_{UML} (i.e. satisfies all invariants from Inv(M_{UML})), then M is a well-formed UML model.

Reading the Standard

Table of Contents	
1. Scope	1
2. Conformance	1
3. Language Use	2
4. Normative References	6
5. Terms and Definitions	10
6. Symbols	10
7. Additional Information	10
8. Normative Annexes	10
9. Informative Annexes	10
10. Bibliography	10
11. Index	10
12. Glossary	10
13. Abbreviations	10
14. Acronyms	10
15. Initialisms	10
16. Symbols	10
17. Part 1 - Structure	21
18. Classes	21

Reading the Standard

Table of Contents	
1. Scope	1
2. Conformance	1
3. Language Use	2
4. Normative References	6
5. Terms and Definitions	10
6. Symbols	10
7. Additional Information	10
8. Normative Annexes	10
9. Informative Annexes	10
10. Bibliography	10
11. Index	10
12. Glossary	10
13. Abbreviations	10
14. Acronyms	10
15. Initialisms	10
16. Symbols	10
17. Part 1 - Structure	21
18. Classes	21

Reading list

Author	Year	Title	Notes
Booch, G.	1991	Object-Oriented Analysis	
Booch, G.	1996	Object-Oriented Design	
Booch, G., Rumbaugh, M., & Jacobson, I.	1999	The Unified Modeling Language User Guide	
Booch, G., Rumbaugh, M., & Jacobson, I.	2005	The Unified Modeling Language	
Booch, G., Rumbaugh, M., & Jacobson, I.	2007	The Unified Modeling Language Supercharge	
Booch, G., Rumbaugh, M., & Jacobson, I.	2013	The Unified Modeling Language, Second Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2015	The Unified Modeling Language, Third Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2017	The Unified Modeling Language, Fourth Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2019	The Unified Modeling Language, Fifth Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2021	The Unified Modeling Language, Sixth Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2023	The Unified Modeling Language, Seventh Edition	
Booch, G., Rumbaugh, M., & Jacobson, I.	2025	The Unified Modeling Language, Eighth Edition	

Meta Object Facility (MOF)

Open Questions...

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

MOF Semantics

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

MOF Semantics

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

MOF Semantics

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

- 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 Odehnik, 2008]

Met-Modelling: (Anticipated) Benefits

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

Benefits for Modelling Tools

- The meta-model *M_{UML}* 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 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. — XML Metadata Interchange (XMI)

Benefits for Modelling Tools Cont'd

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

- 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.

- 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.

- 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 Gk::Button, Gk::Toolbar, etc. **corresponding** to the stereotype.

- 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 Gk::Button, Gk::Toolbar, etc. **corresponding** to the stereotype.
- **Et voilà:** we can model Gk-GUIs and generate code for them.

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

- 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 GK-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 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* for 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-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!

- 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 GK-UML model into a **UML model**, where the inheritance from classes like GK:Button is made explicit: The transformation would add this class GK:Button and the inheritance relation and remove the stereotype.

- 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 GK-UML model into a **UML model**, where the inheritance from classes like GK:Button is made explicit: The transformation would add this class GK:Button and the inheritance relation and remove the stereotype.
- Similarly, one could have a **GUI-UML** model transformed into a **GK-UML** model, or a **Q-UML** model.
- The former a PIM (Platform Independent Model), the latter a PSM (Platform Specific Model) — cf. MDA.

- 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.

- 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**.

```

<?xml version="1.0" encoding="utf-8" ?>
<XML meta:version="1.2" xmlns:uml="http://www.omg.org/spec/UML/2007/09/01/UML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.omg.org/spec/UML/2007/09/01/UML http://www.omg.org/spec/UML/2007/09/01/UML.xsd">
  <diagram name="Smool" <!-- UML diagram -->
    <class name="Smool" <!-- UML class -->
      <association name="Controlled" <!-- UML association -->
        <end class="Smool" <!-- UML class -->
        <end class="Condition" <!-- UML class -->
      </association>
      <association name="Controlled" <!-- UML association -->
        <end class="Smool" <!-- UML class -->
        <end class="update" <!-- UML class -->
      </association>
      <association name="Controlled" <!-- UML association -->
        <end class="update" <!-- UML class -->
        <end class="USA" <!-- UML class -->
      </association>
    </class>
  </diagram>
</XML>
  
```

References

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

[Crane and Dingel 2007] Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. ad-hoc: statecharts, not all models are created equal. *Software and Systems Modeling*, 6(4):419-435.

[Frieder and Wehrlein, 2004] Frieder, C. and Wehrlein, H. (2004). Behavioral subtyping relations for statecharts. In *Proc. 1st IEEE Int'l Workshop UML and Formal Methods*, number 1818 in Lecture Notes in Computer Science. Springer-Verlag.

[Hand and Gery, 1997] Hand, D. and Gery, E. (1997). Executable object modeling with statecharts. *IEEE Computer*, 30(7):31-42.

[Lissov, 1988] Lissov, B. (1988). Data abstraction and hierarchy. *SIGPLAN Not.*, 23(9):17-34.

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

[OMG, 2003] OMG (2003). Uml 2.0 proposal. <http://www.omg.org/uml2/uml2.html>.

[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 Volter, 2009] Stahl, T. and Volter, M. (2009). Modellgetriebene Softwareentwicklung. Openix-Verlag, Heidelberg.