### Contents & Goals

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

**This Lecture:**
- Educational Objectives:
  - Capabilities for following tasks/questions.
  - What's the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the uplink semantics of inheritance?
  - What's the effect of inheritance on LSCs, State Machines, System States?
  - What's the idea of Meta-Modelling?

**Content:**
- Quickly: Behavioural Features, Active vs. Passive
- Inheritance in UML: concretesyntax
- Liskov Substitution Principle—desired semantics
- Two approaches to obtain desired semantics
- The UML MetaModel

### Active and Passive 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.

### Active and Passive Objects: Nomenclature

[Hareland 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 systems 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?

<table>
<thead>
<tr>
<th>Active</th>
<th>Passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

### Passive and Reactive

- So why don't we understand passive/reactive?
- Assume passive objects \( u_1 \) and \( u_2 \), and active object \( u \), 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 requiring that reactive implies active for model well-formedness.
- Requiring for model well-formedness that events are never sent to instances of non-reactive classes.

### Further Reading

[Hareland Gery, 1997] Delegate all dispatching of events to the active objects.
Yet we could explain pre/post in OCL (if we wanted to).

Forsimplicity, weleavethenotion of stepsuntouched, we constructour

• Transitionactionsmayfill inthereturnvalue.

• The class' behaviouralfeature can be provided by:

E

⟩ ⟩

signal

• In UML, the former is

• In C++ lingo: distinguish

• The activeobject

• If an objectis an activeclass, then

• Dispatching an event

• Association

• Send to
dest's

• Create an instance

• Establish that of each signal we

• Observation: the ether only has

• Where an ether

• In the current setting, the (local) state of objects is only modified by

• In a setting with Java as action language: operation is a method body.

• Function composition of transformers (clear but tedious: non-termination).

• In general, there are also

• And what about methods?

• Behavioural Features: Visibility and Properties
State Machines: Discussion.

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 child 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 "semantic 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)
  - None is the subset of another (i.e., for each pair of communities exist pictures meaning different things)

Optimistic view

- Toolsexist with complete and consistent code generation.

Recall: Abstract Syntax

Recall: A signature (with signals) is a tuple

Now (finally): extend to

where $F/\triangleright$ are methods, analogously to attributes and $\subseteq (\mathcal{B} \times \mathcal{B}) \cup (\mathcal{B}^+ \times \mathcal{B}^+)$ is a generalisation such that $C \triangleright D$ for no $C \in \mathcal{B}^+$ ("acyclic"). $C \triangleright D$ reads as

- $C$ is a generalisation of $D$,
- $D$ is a specialisation of $C$,
- $D$ inherits from $C$,
- $D$ is a sub-class of $C$,
- $C$ is a super-class of $D$,
- ...

Inheritance: Syntax

- Recall: Abstract Syntax
  - A signature (with signals) is a tuple
  - Now (finally): extend to
  - $\triangleright$ are methods, analogously to attributes and $\subseteq (\mathcal{B} \times \mathcal{B}) \cup (\mathcal{B}^+ \times \mathcal{B}^+)$ is a generalisation such that $C \triangleright D$ for no $C \in \mathcal{B}^+$ ("acyclic"). $C \triangleright D$ reads as
    - $C$ is a generalisation of $D$,
    - $D$ is a specialisation of $C$,
    - $D$ inherits from $C$,
    - $D$ is a sub-class of $C$,
    - $C$ is a super-class of $D$,
Recall: Reflexive, Transitive Closure of Generalisation

Definition.

Given classes $C_0, C_1, D \in \mathcal{BV}$, we say $D$ inherits from $C_0$ via $C_1$ if and only if there are $C_1_0, \ldots, C_n_0, C_1_1, \ldots, C_m_1 \in \mathcal{BV}$ such that $C_0 \triangleright C_1_0 \triangleright \ldots C_n_0 \triangleright C_1_1 \triangleright \ldots C_m_1 \triangleright D$.

We use ‘$\triangleright$’ to denote the reflexive, transitive closure of ‘$	riangleright$’.

In the following, we assume

- that all attribute (method) names are of the form $C::v, C \in \mathcal{BV} \cup \mathcal{BX}$ ($C::f, C \in \mathcal{BV}$),
- that we have $C::v \in \text{atr}(C)$ resp. $C::f \in \text{mth}(C)$ if and only if $v(f)$ appears in an attribute (method) compartment of $C$ in a class diagram.

We still want to accept "context $C$ in $v_0$", which $v$ is meant? Later!

Desired Semantics of Specialisation: Subtyping

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

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].

(\textit{Liskov Substitution Principle (LSP).})

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

S sub-type of T: $\iff \forall o_1 \in S \exists o_2 \in T \forall P T : \text{context } C o_1 \triangleq P T \Rightarrow C o_2"$.

In other words: [Fischer and Wehrheim, 2000]

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

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

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

- Narrow interpretation: another object in the model.
- Wide interpretation: another modeler.

**C**x : Int

\[ \text{Int} \] D – 20 – 2014 -02-03 – Ssem –

**Motivations for Generalisation**

- Re-use,
- Sharing,
- Avoiding redundancy,
- Modularisation,
- Separation of concerns,
- Abstraction,
- Extensibility,
- ...

→ See textbook on object-oriented analysis, development, programming.

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

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

Given:

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

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

\[ \langle \langle \text{signal} \rangle \rangle E \langle \langle \text{signal} \rangle \rangle F \]

Wanted:

- \( x > 0 \) also well-typed for \( D_1 \)
- assignment \( \text{itsC}_1 := \text{itsD}_1 \) being well-typed
- \( \text{itsC}_1.x = 0, \text{itsC}_1.f(0), \text{itsC}_1!F \) being well-typed (and doing the right thing).

Approach:

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

Static Typing Cont'd

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

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

\[ \langle \langle \text{signal} \rangle \rangle E \langle \langle \text{signal} \rangle \rangle F \]

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 in the Standard and Programming Lang.

- In the standard, Section 11.3.10, "Call Operation Action": "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) compiletime 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?
Late Binding in the Standard and Programming Languages

- 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."

- In C++,
  
  methods are by default "(early) compiletime 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 realize 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,

  - we (by definition) only see and change the \( C \)-part.

  - We cannot tell whether \( u \) is a \( C \) or a \( D \) instance.

  - So we immediately also have behavioural/dynamic subtyping.

**Difficult: Dynamic Subtyping**

\[
\begin{align*}
C \ f(\text{Int}) : \text{Int} \\
D \ f(\text{Int}) : \text{Int}
\end{align*}
\]

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

- Examples (C++):

  \[
  \begin{align*}
  \text{int } C :: f(\text{int}) &= \{ \text{return } 0; \} \\
  \text{int } D :: f(\text{int}) &= \{ \text{return } 1; \}
  \end{align*}
  \]

  vs.

  \[
  \begin{align*}
  \text{int } C :: f(\text{int}) &= \{ \text{return } \text{(rand()) % 2}; \} \\
  \text{int } D :: f(\text{int} \ x) &= \{ \text{return } \text{(x % 2); \}
  \end{align*}
  \]

**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]
Ensuring Sub-Typing for State Machines

In the standard, Section 7.3.36, "Operations:"

- In an interface hierarchy, an initial state cannot be defined for a state machine that is inherited from a base state machine.
- In an interface hierarchy, an initial state cannot be defined for a state machine that is inherited from a base state machine.
- In an interface hierarchy, an initial state cannot be defined for a state machine that is inherited from a base state machine.
- In an interface hierarchy, an initial state cannot be defined for a state machine that is inherited from a base state machine.

Attach a transition to a different target (limited).

Attach a transition to a different target (limited).

Add more states.

Add more states.

Add things into (hierarchical) states.

Add things into (hierarchical) states.

Roughly (cf. UserGuide, p. 760, for details), "has a stronger post-condition." (3)

"Has a weaker pre-condition," (3)

Relax to "fewer observable behaviour", thus admitting the sub-type to do more work on inputs.

One could, e.g., want to consider execution time.

And not necessarily the end of the story:

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

Note (ii) on the semantic variation points:

- Operation (i) accepts fewer input values, has fewer behaviour, invariance.
- Operation (i) accepts fewer input values, has fewer behaviour, invariance.
- Operation (i) accepts fewer input values, has fewer behaviour, invariance.
- Operation (i) accepts fewer input values, has fewer behaviour, invariance.

Determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.

Semantic Variation Points

Sub-Typing Principles Cont’d

In the standard, Section 7.3.36, "Operations:"

- We consider multiple classes in an inheritance hierarchy can have state machines.

In the CASE tool we consider, multiple classes are running.

Technically, the idea is that (by late binding) only the statemachine of the most specialised class is running. (But method implementations can still destroy that property.)

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

Sets to a copy of the original one.

Instead, the statemachine of a sub-class can only be obtained by drawing from scratch.

Note that the statemachine of a sub-class cannot be inherited from a base state machine.

Note that the statemachine of a sub-class cannot be inherited from a base state machine.

Note that the statemachine of a sub-class cannot be inherited from a base state machine.
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.

– 20 – 2014-02-03 – Subtyping –

We’ll discuss two approaches to semantics:

• Domain-inclusion Semantics (more theoretical)
• Uplink Semantics (more technical)

Domain-inclusion Semantics

Let $(\mathcal{P}, \mathcal{C}, \mathcal{B}, V, \mathcal{A}, \mathcal{F}, \triangleright)$ be a signature.

Now a structure $\mathcal{W}$:

• as before, maps to 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. $\forall C \in \mathcal{B}: \mathcal{W}(C) \supseteq \bigcup \mathcal{C} \triangleright D \mathcal{W}(D)$.

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

– 20 – 2014-02-03 – Subtyping –

Domain-inclusion System States

Now: a system state of $\mathcal{P}$ wrt. $\mathcal{W}$ is a type-consistent mapping $\sigma: \mathcal{W}(\mathcal{B}) \mapsto \mathcal{V} \mapsto \mathcal{B}(\mathcal{C}) \cup \mathcal{W}(\mathcal{B}_0,1) \cup \mathcal{W}(\mathcal{B}^*)$.

• as before, $\sigma(u)(v) \in \mathcal{W}(\tau)$ if $v: \tau$, $\tau \in \mathcal{C}$ or $\tau \in \{ C^*, C_0, 1 \}$.
• changed, dom($\sigma(u)$) = $\bigcup C_0 \preceq C \mathcal{A}(C_0)$.

Example: $C_x : \text{Int}$, $D_x : \text{Int}$, $y : \text{Int}$, $n_0, 1$.

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

– 20 – 2014-02-03 – Subtyping –

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 $\triangleright$."
Example C \[\vdash \] : A,D 

We propose several new typing rules:

- \[\vdash \] : A,D 

- \[\vdash \] : A,D 

- \[\vdash \] : A,D 

More interesting: Well-Typed-ness with Visibility Cont’d

Well-Typed-ness with Visibility Cont’d
\[ C \text{-instantiation also binds to: } C \in \mathcal{E} \]

- **Uplink Semantics**

  - **Domain Inclusion and Interactions**

  - **Transformers (Domain Inclusion)**
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 $f$) in the context of class $D$,

• let $C$ be the smallest class wrt. "$\preceq$" such that

• $C \preceq D$, and

• $C :: v \in \text{atr}(D)$

• then there exists (by definition) $C \triangleleft C_1 \triangleleft \ldots \triangleleft C_n \triangleleft D$,

• normalise $v$ to (replace by) $\text{uplink} C_n \rightarrow \ldots \rightarrow \text{uplink} C_1$. $C :: v$

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

Definition of structure remains unchanged.

• Definition of system state remains unchanged.

• Typing and transformers remain unchanged — the pre-processing has put everything in shape.

Satisfying OCL Constraints (Uplink)

• Let $M = (BV/BW, C_7/BW, CB/C_5, C_1)$ be a UML model, and $BW$ a structure.

• We (continuously) say $M |_{C} = expr$ for context $C$ in $v$:

• $\forall \pi = (\sigma_i)_{i \in N} \in C_2 \land C_3 \forall i \in N \forall u \in \text{dom}(\sigma_i) \cap BW(C)$:

• $I_{C_2} \text{expr}_0 /_{C_3} (\sigma_i, \{\text{self} \mapsto u\}) = \text{true}$.

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

Transformers (Uplink)

• What has to change is the create transformer: $\text{create}(C, expr, v)$

• Assume, $C$'s inheritance relations are as follows.

• $C_1, 1 \triangleleft \ldots \triangleleft C_1, n \triangleleft C, \ldots \triangleleft C_m, 1 \triangleleft \ldots \triangleleft C_m, n \triangleleft C$.

• Then, we have to

• create one fresh object for each part, e.g. $u_1, 1, \ldots, u_1, n, \ldots, u_m, 1, \ldots, u_m, n$,

• setup the uplinks recursively, e.g. $\sigma(u_1, 2) \cdot \text{uplink} C_1, 1 = u_1, 1$.

• And, if we had constructors, be careful with their order.
As long as we're early binding?

theory

pre-processing

•

if it can be approached soodramatically

conciseness

And it also •

•

expressive power.

meta-modelling: idea and example

what's the point of

exercise

domain inclusion vs. uplink semantics: motives

note:

\[ d \)

cp -> mostspec -> uplink

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]

\[ \rightarrow \]
Meta-Modelling: Why and What

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

The idea is simple:

- If a modelling language is about modelling things,
- and if UML models are and comprise things,
- then why not model those in a modelling language?

In other words: Why not have a model $\mathcal{M}$ such that the set of legal instances of $\mathcal{M}$ is the set of well-formed (!) UML models.

---

Meta-Modelling: Example

For example, let's consider a class.

- A class has (on a superficial level)
- a name,
- any number of attributes,
- any number of behavioural features.

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

Behavioural features in addition have a boolean attribute isQuery, any number of parameters, and 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

---

Classes

[OMG, 2007b, 32]

Figure 7.12 - Classes diagram of the Kernel package
Meta-modelling has already been used for UML 1.x. For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: Infrastructure and Superstructure.

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

Core UML

MOF

CWM Profiles

Figure 0-1 Overview of architecture

- Class, Object
- Action, Filmstrip
- Package, Snapshot
- Class, State, Transition, Flow, ...

Superstructure (concrete syntax)

- ClassBox, StateBox, TransitionLine, ...

Superstructure (abstract syntax)

Infrastructure (with semantics)

- Diagram
- Interchange
- Node, Edge...

Interesting: Declaration/Definition

Figure 13.6 Common Behavior

UML Architecture

[OMG, 2003, 8]

- Meta-modelling has already been used for UML 1.x.
- For UML 2.0, the request for proposals (RFP) asked for a separation of concerns: Infrastructure and Superstructure.
- One reason: sharing with MOF (see later) and, e.g., CWM.
Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

- CommonBehaviors
- UseCases
- Classes
- StateMachines
- Interactions
- CompositeStructures
- Components
- Deployments
- AuxiliaryConstructs
- Activities
- Actions

Modelling vs. Meta-Modelling

\[ C_v : Z_{CB} = (\{ Z \}, \{ C \}, \{ v \}, \{ C \rightarrow v \} ) \]

Model \((M1)\)

\[ \sigma = \{ u \rightarrow \{ v \rightarrow 0 \} \} \in \text{Model}(M1) \]

Instance \((M0)\)

Class
Property
Type

\[ C_v : Z_{CB} = (\{ Z \}, \{ C \}, \{ v \}, \{ C \rightarrow v \} ) \]

\[ \sigma = \{ u \rightarrow \{ v \rightarrow 0 \} \} \in \text{Model}(M1) \]

Instance \((M0)\)

Meta-Model \((M2)\)

Model \((M1)\)

Instance \((M0)\)
Well-formedness as Constraints in the Meta-Model

Well-formedness as Constraints in the Meta-Model
Specifies each feature defined in the classifier. Subsets

- Feature : Feature [*]

Classifier::feature

Classifier::allFeatures():

• attribute: Property [*]

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a

Additional Operations

Begin attribute and operation names with a lowercase letter.

with an uppercase character).

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

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

[5] The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the

relationships). Default value is

Attribute names typically begin with a lowercase letter. Multi-word names are often formed by concatenating the words

and using lowercase for all letters except for upcasing the first letter of each word but the first.

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

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to

redefined by classifiers that have different specialization constraints.

The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example,

The query inheritableMembers() gives all of the members of a classifier that may be inherited in one of its descendants,


ClassA::height is an attribute of type Integer with a default initial value of 5.

ClassA::shape is an attribute with type Rectangle.

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

compartments separated by horizontal lines containing features or other members of the classifier. The specific type of

RedefinableElement::redefinedElement

RedefinableElement::redefinedElement

Window

Display

private

display()

private

attachX(xWin: XWindow)

private

xWin: XWindow

Area

Shape

Display

private

display()

private

attachX(xWin: XWindow)

private

xWin: XWindow

Area

Shape

Display
• not only graphical representations (that are graphs) then are Object diagrams (that are graph based, "is-instance-of" relation.

Definea Otherapproach:

• (For instance, MOF doesn’t have a notion of Signal, oursignature has.) signatures foreachlevel. This is (ineffect) theright direction, but may require new (orextended) oursignature-based theory

This is (ineffect) theright direction, but may require new (orextended) oursignature-based theory

M3are

•/ general : Classifier[*]
Classifier::parents(): Set(Classifier);


M2are

•/ attribute: Property[*]
Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets

M0are objectdiagrams/systemstates

•/ size
A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is itself nor may its instances also be its subclasses.

Notation

• left justify attributes and operations in plain face.

With an uppercase character).

For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e, begin them with an uppercase character).

Center the name of the classifier in boldface.

Center keyword (including stereotype names) in plain face within guillemets above the classifier name.

Semantics

•/ classification
A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization in the model.

Examples

•/ substitution : Substitution

The notion of power type was inspired by the notion of power set. A power set is defined as a set whose instances are

Examples in the GeneralizationSet sub clause, below.)

Account, then, are instances of the power type: Bank Account Type. In other words, Checking Account and Savings

Package PowerTypes

‘RedefinableElement (from Kernel)” on page 130

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

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

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

JOM, 2007a

Display()
One approach:
• Treat it with our signature-based theory
  This is (ineffect) the right direction, but may require new (or extended) signatures for each level.
  (For instance, MOF doesn’t have an 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 clearly still research, e.g.
  [Buschermohle and Oelerink, 2008]

Benefits: Overview
• We’ll (superficially) look at three aspects:
  • Benefits for Modelling Tools.
  • Benefits for Language Design.
  • Benefits for Code Generation and MDA.

Benefits for Modelling Tools
• The meta-model $M$ 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 (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 oftentimes 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)
• Note: Apriori, 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: Apriori, 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 XMI standard. And different tools by different vendors often seem to lie at opposite ends of the scale of interpretation. Which is surely a coincidence. In some cases, it's possible to fix things with, e.g., XSLT scripts, but full vendor independence is today not given.

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

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

Benefits: Overview

• We'll (superficially) look at three aspects:
  • Benefits for Modeling Tools.
  ✔
  • Benefits for Language Design.
  • Benefits for Code Generation and MDA.
• 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. [Stahland Völter, 2005].)


Benefits for Language Design Cont'd

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

→ a new language standing next to UML, CWM, etc.

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

But we can use all tools for MOF (or MOF-like things). For instance, Eclipse EMF/GMF/GEF.


Benefits: Overview

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


Benefits for Model (to Model) Transformation

• There are manifold applications for model-to-model transformations:
  • For instance, toolsupport for refactorings, like moving common attributes upward 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.
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 situations where a graphical and abstract representation of something is desired which has a clear and direct mapping to some textual representation."

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

Example: Model and XMI

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