Software Design, Modelling and Analysis in UML

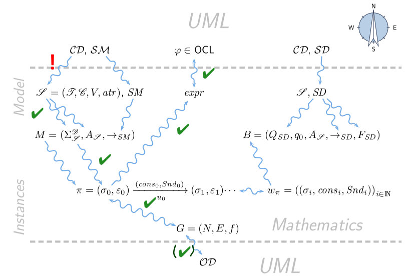
Lecture 05: Class Diagrams I

2012-11-07

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

Albert-Ludwigs-Universität Freiburg, Germany

Course Map



Contents & Goals

Last Lecture:

- OCL Semantics
- Object Diagrams

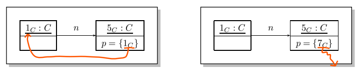
This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is a class diagram?
  - For what purposes are class diagrams useful?
  - Could you please map this class diagram to a signature?
  - Could you please map this signature to a class diagram?
- **Content:**
  - Object Diagrams Cont'd.
  - Study UML syntax.
  - Prepare (extend) definition of signature.
  - Map class diagram to (extended) signature.
  - Stereotypes – for documentation.

Recall: Corner Cases

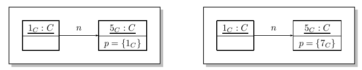
Closed Object Diagrams vs. Dangling References

Find the 10 differences! (Both diagrams shall be complete.)



Closed Object Diagrams vs. Dangling References

Find the 10 differences! (Both diagrams shall be complete.)



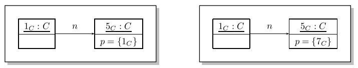
Definition. Let σ be a system state. We say attribute $v \in V_{0,1,*}$ has a dangling reference in object $u \in \mathrm{dom}(\sigma)$ if and only if the attribute's value comprises an object which is not alive in σ, i.e. if

$$\sigma(u)(v) \not\subseteq \mathrm{dom}(\sigma).$$

We call σ closed if and only if no attribute has a dangling reference in any object alive in σ.

Closed Object Diagrams vs. Dangling References

Find the 10 differences! (Both diagrams shall be complete.)



Definition. Let σ be a system state. We say attribute $v \in V_{0,1,*}$ has a dangling reference in object $u \in \mathrm{dom}(\sigma)$ if and only if the attribute's value comprises an object which is not alive in σ, i.e. if

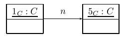
$$\sigma(u)(v) \not\subseteq \mathrm{dom}(\sigma).$$

We call σ closed if and only if no attribute has a dangling reference in any object alive in σ.

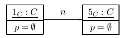
Observation: Let G be the (!) complete object diagram of a **closed** system state σ. Then the nodes in G are labelled with $\mathscr{T}$-typed attribute/value pairs only.

Special Notation

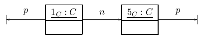
- $\mathscr{S} = (\{Int\}, \{C\}, \{n, p : C_*\}, \{C \mapsto \{n, p\}\})$.
- Instead of



we want to write



or

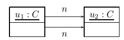


to **explicitly** indicate that attribute $p : C_*$ has value $\emptyset$ (also for $p : C_{0,1}$).

Aftermath

We slightly deviate from the standard (for reasons):

- In the course, $C_{0,1}$ and C_*-typed attributes **only** have **sets as values**. UML also considers multisets, that is, they can have



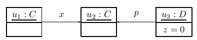
(This is not an object diagram in the sense of our definition because of the requirement on the edges E. Extension is straightforward but tedious.)

- We **allow** to give the valuation of $C_{0,1}$- or C_*-typed attributes in the **values compartment**.
  - Allows us to indicate that a certain r is not referring to another object.
  - Allows us to represent "dangling references", i.e. references to objects which are not alive in the current system state.
- We introduce a graphical representation of $\emptyset$ values.

The Other Way Round

The Other Way Round

- If we **only** have a picture as below, we typically assume that it's **meant to be** an object diagram wrt. **some** signature and structure.

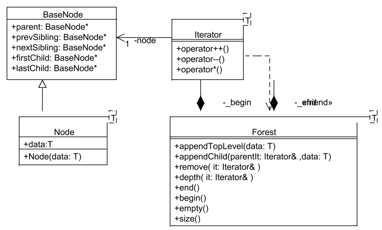


- In the example, we can conclude (by **"good will"**) that the author is referring to **some** signature $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ with at least
  - $\{C, D\} \subseteq \mathscr{C}$
  - $T \in \mathscr{T}$
  - $\{x : C_*, p : C_*, z : T\} \subseteq V$
  - $\{x\} \subseteq atr(C)$
  - $\{p, z\} \subseteq atr(D)$

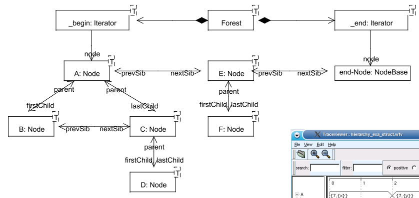
  and a structure with
  - $\{u_1, u_2\} \subseteq \mathscr{D}(C)$
  - $\{u_3\} \subseteq \mathscr{D}(D)$
  - $0 \in \mathscr{D}(T)$

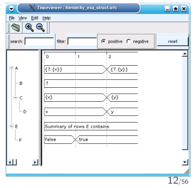
Example: Object Diagrams for Documentation

Example: Data Structure [Schumann et al., 2008]



Example: Illustrative Object Diagram [Schumann et al., 2008]





OCL Consistency

OCL Satisfaction Relation

In the following, $\mathscr{S}$ denotes a signature and $\mathscr{D}$ a structure of $\mathscr{S}$.

Definition (Satisfaction Relation).
Let φ be an OCL constraint over $\mathscr{S}$ and $\sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}}$ a system state.
We write

- $\sigma \models \varphi$ ("σ **satisfies** φ") if and only if $I[\![\varphi]\!](\sigma, \emptyset) = $ *true*.
- $\sigma \not\models \varphi$ if and only if $I[\![\varphi]\!](\sigma, \emptyset) = $ *false*.

Note: In general we **can't** conclude from $\neg(\sigma \models \varphi)$ to $\sigma \not\models \varphi$ or vice versa.

Object Diagrams and OCL

- Let G be an object diagram of signature $\mathscr{S}$ wrt. structure $\mathscr{D}$.
  Let $expr$ be an OCL expression over $\mathscr{S}$.

  We say G **satisfies** $expr$, denoted by $G \models expr$, if and only if

  $$\forall\, \sigma \in G^{-1} : \sigma \models expr.$$

- If G is **complete**, we can also talk about "$\not\models$".

  (Otherwise, to avoid confusion, avoid "$\not\models$": G^{-1} could comprise system states in which $expr$ evaluates to *true*, *false*, and $\bot$.)

Object Diagrams and OCL

- Let G be an object diagram of signature $\mathscr{S}$ wrt. structure $\mathscr{D}$.
  Let $expr$ be an OCL expression over $\mathscr{S}$.

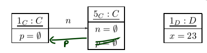
  We say G **satisfies** $expr$, denoted by $G \models expr$, if and only if

  $$\forall\, \sigma \in G^{-1} : \sigma \models expr.$$

- If G is **complete**, we can also talk about "$\not\models$".

  (Otherwise, to avoid confusion, avoid "$\not\models$": G^{-1} could comprise system states in which $expr$ evaluates to *true*, *false*, and $\bot$.)

- **Example**: (complete — what if not complete wrt. object/attribute/both?)



  - context C inv : n -> isEmpty()
  - context C inv : $p.n$ -> isEmpty()
  - context D inv : $x \neq 0$

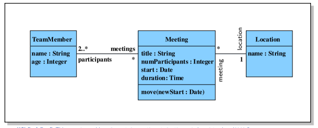
OCL Consistency

Definition (Consistency). A set $Inv = \{\varphi_1, \dots, \varphi_n\}$ of OCL constraints over $\mathscr{S}$ is called consistent (or satisfiable) if and only if there exists a system state of $\mathscr{S}$ wrt. $\mathscr{D}$ which satisfies all of them, i.e. if

$$\exists \sigma \in \Sigma_{\mathscr{S}}^{\mathscr{D}} : \sigma \models \varphi_1 \wedge \ldots \wedge \sigma \models \varphi_n$$

and inconsistent (or unrealizable) otherwise.

OCL Inconsistency Example



((C) Prof. Dr. P. Thiemann, http://proglang.informatik.uni-freiburg.de/teaching/swt/2008/)

- context $Location$ inv :
  $name$ = 'Lobby' implies $meeting$ -> $isEmpty()$
- context $Meeting$ inv :
  $title$ = 'Reception' implies $location . name = "Lobby"$
- allInstances$_{Meeting}$ -> exists($w : Meeting \mid w . title$ = 'Reception')

Deciding OCL Consistency

- Whether a set of OCL constraints is satisfiable or not is **in general not as obvious** as in the made-up example.
- **Wanted**: A procedure which decides the OCL satisfiability problem.

Deciding OCL Consistency

- Whether a set of OCL constraints is satisfiable or not is **in general not as obvious** as in the made-up example.
- **Wanted**: A procedure which decides the OCL satisfiability problem.
- **Unfortunately**: in general **undecidable**.

  Otherwise we could, for instance, solve **diophantine equations**

  $$c_1 x_1^{n_1} + \cdots + c_m x_m^{n_m} = d.$$

Deciding OCL Consistency

- Whether a set of OCL constraints is satisfiable or not is **in general not as obvious** as in the made-up example.
- **Wanted**: A procedure which decides the OCL satisfiability problem.
- **Unfortunately**: in general **undecidable**.

  Otherwise we could, for instance, solve **diophantine equations**

  $$c_1 x_1^{n_1} + \cdots + c_m x_m^{n_m} = d.$$

  Encoding in OCL:

  allInstances$_C$ -> exists($w : C \mid c_1 * w.x_1^{n_1} + \cdots + c_m * w.x_m^{n_m} = d$).

Deciding OCL Consistency

- Whether a set of OCL constraints is satisfiable or not is **in general not as obvious** as in the made-up example.
- **Wanted**: A procedure which decides the OCL satisfiability problem.
- **Unfortunately**: in general **undecidable**.

  Otherwise we could, for instance, solve **diophantine equations**

  $$c_1 x_1^{n_1} + \cdots + c_m x_m^{n_m} = d.$$

  Encoding in OCL:

  allInstances$_C$ -> exists($w : C \mid c_1 * w.x_1^{n_1} + \cdots + c_m * w.x_m^{n_m} = d$).

- **And now**? Options: [Cabot and Clarisó, 2008]
  - Constrain OCL, use a **less rich** fragment of OCL.
  - Revert to **finite domains** — basic types vs. number of objects.

OCL Critique

- **Expressive Power**:
  - "Pure OCL expressions only compute primitive recursive functions, but not recursive functions in general." [Cengarle and Knapp, 2001]
  - **Evolution over Time**: "finally $self.x > 0$"

    Proposals for fixes e.g. [Flake and Müller, 2003]. (Or: sequence diagrams.)
  - **Real-Time**: "Objects respond within 10s"

    Proposals for fixes e.g. [Cengarle and Knapp, 2002]
  - **Reachability**: "After insert operation, node shall be reachable."

    Fix: add transitive closure.

OCL Critique

- **Expressive Power**:
  - "Pure OCL expressions only compute primitive recursive functions, but not recursive functions in general." [Cengarle and Knapp, 2001]
  - **Evolution over Time**: "finally $self.x > 0$"

    Proposals for fixes e.g. [Flake and Müller, 2003]. (Or: sequence diagrams.)
  - **Real-Time**: "Objects respond within 10s"

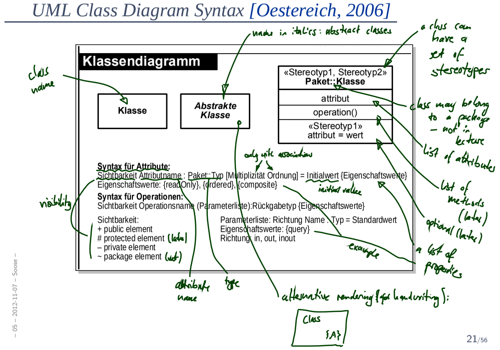
    Proposals for fixes e.g. [Cengarle and Knapp, 2002]
  - **Reachability**: "After insert operation, node shall be reachable."

    Fix: add transitive closure.
- **Concrete Syntax**

  "The syntax of OCL has been criticized – e.g., by the authors of Catalysis [...] – for being hard to read and write.
  - OCL's expressions are stacked in the style of Smalltalk, which makes it hard to see the scope of quantified variables.
  - Navigations are applied to atoms and not sets of atoms, although there is a collect operation that maps a function over a set.
  - Attributes, [...], are partial functions in OCL, and result in expressions with undefined value." [Jackson, 2002]

UML Class Diagrams: Stocktaking

UML Class Diagram Syntax [Oestereich, 2006]



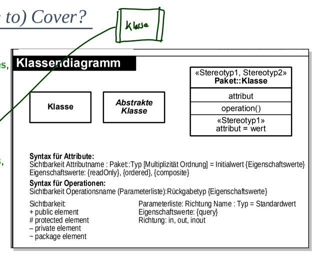
What Do We (Have to) Cover?

A **class**

- has a set of **stereotypes**,
- has a **name**,
- belongs to a **package**,
- can be **abstract**,
- can be **active**,
- has a set of **operations**,
- has a set of **attributes**.

Each **attribute** has

- a **visibility**,
- a **name**, a **type**,
- a **multiplicity**, an **order**,
- an **initial value**, and
- a set of **properties**, such as **readOnly**, **ordered**, etc.



Wanted: places in the signature to represent the information from the picture.

Extended Signature

Recall: Signature

$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- (basic) types $\mathscr{T}$ and classes $\mathscr{C}$, (both finite),
- typed attributes V, τ from $\mathscr{T}$ or $C_{0,1}$ or C_*, $C \in \mathscr{C}$,
- $atr : \mathscr{C} \to 2^V$ mapping classes to attributes.

Too abstract to represent class diagram, e.g. no "place" to put class **stereotypes** or attribute **visibility**.

So: **Extend** definition for classes and attributes: Just as attributes already have types, we will assume that

- classes have (among other things) **stereotypes** and
- attributes have (in addition to a type and other things) a **visibility**.

Extended Classes

From now on, we assume that each class $C \in \mathscr{C}$ has:

- a finite (possibly empty) set S_C of **stereotypes**,
- a boolean flag $a \in \mathbb{B}$ indicating whether C is **abstract**,
- a boolean flag $t \in \mathbb{B}$ indicating whether C is **active**.

We use $S_{\mathscr{C}}$ to denote the set $\bigcup_{C \in \mathscr{C}} S_C$ of stereotypes in $\mathscr{S}$.
(Alternatively, we could add a set St as 5-th component to $\mathscr{S}$ to provides the stereotypes (names of stereotypes) to choose from. But: too unimportant to care.)

Extended Classes

From now on, we assume that each class $C \in \mathscr{C}$ has:

- a finite (possibly empty) set S_C of **stereotypes**,
- a boolean flag $a \in \mathbb{B}$ indicating whether C is **abstract**,
- a boolean flag $t \in \mathbb{B}$ indicating whether C is **active**.

We use $S_{\mathscr{C}}$ to denote the set $\bigcup_{C \in \mathscr{C}} S_C$ of stereotypes in $\mathscr{S}$.
(Alternatively, we could add a set St as 5-th component to $\mathscr{S}$ to provides the stereotypes (names of stereotypes) to choose from. But: too unimportant to care.)

Convention:

- We write
$$\langle C, S_C, a, t \rangle \in \mathscr{C}$$
when we want to refer to all aspects of C.
- If the new aspects are irrelevant (for a given context), we simply write $C \in \mathscr{C}$ i.e. old definitions are still valid.

Extended Attributes

- From now on, we assume that each attribute $v \in V$ has (in addition to the type):
  - a **visibility**
$$\xi \in \{\underbrace{\text{public}}_{:=+}, \underbrace{\text{private}}_{:=-}, \underbrace{\text{protected}}_{:=\#}, \underbrace{\text{package}}_{:=\sim}\}$$
  - an **initial value** $expr_0$ given as a word from **language for initial values**, e.g. OCL expresions.
(If using Java as **action language** (later) Java expressions would be fine.)
  - a finite (possibly empty) set of **properties** P_v.
We define $P_{\mathscr{C}}$ analogously to stereotypes.

Extended Attributes

- From now on, we assume that each attribute $v \in V$ has (in addition to the type):
  - a **visibility**

$$\xi \in \{\underbrace{\text{public}}_{:=+}, \underbrace{\text{private}}_{:=-}, \underbrace{\text{protected}}_{:=\#}, \underbrace{\text{package}}_{:=\sim}\}$$

  - an **initial value** $expr_0$ given as a word from **language for initial values**, e.g. OCL expresions.

    (If using Java as **action language** (later) Java expressions would be fine.)
  - a finite (possibly empty) set of **properties** P_v.

    We define $P_{\mathscr{C}}$ analogously to stereotypes.

Convention:

- We write $\langle v : \tau, \xi, expr_0, P_v \rangle \in V$ when we want to refer to all aspects of v.
- Write only $v : \tau$ or v if details are irrelevant.

And?

- **Note**:
  All definitions we have up to now **principally still apply** as they are stated in terms of, e.g., $C \in \mathscr{C}$ — which still has a meaning with the extended view.

  For instance, system states and object diagrams remain mostly unchanged.
- **The other way round**: **most** of the newly added aspects **don't contribute** to the constitution of system states or object diagrams.

And?

- **Note:**
  All definitions we have up to now **principally still apply** as they are stated in terms of, e.g., $C \in \mathscr{C}$ — which still has a meaning with the extended view.

  For instance, system states and object diagrams remain mostly unchanged.

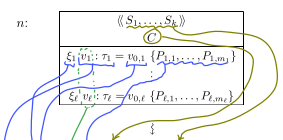
- **The other way round:** **most** of the newly added aspects **don't contribute** to the constitution of system states or object diagrams.

- Then what **are** they useful for...?
- First of all, to represent class diagrams.
- And then we'll see.

Mapping UML CDs to Extended Signatures

From Class Boxes to Extended Signatures

A class box n **induces** an (extended) signature class as follows:



$$\mathscr{C}(n) := \langle C, \{S_1, \dots, S_k\}, a(n), t(n)\rangle$$

$$V(n) := \{\langle v_1 : \tau_1, \xi_1, v_{0,1}, \{P_{1,1}, \dots, P_{1,m_1}\}\rangle, \dots, \langle v_\ell : \tau_\ell, \xi_\ell, v_{0,\ell}, \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}\rangle\}$$

$$atr(n) := \{C \mapsto \{v_1, \dots, v_\ell\}\}$$

where

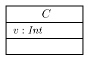
- "abstract" is determined by the font:

  $a(n) = \begin{cases} true & \text{, if } n = \boxed{C} \text{ or } n = \boxed{C_{\{A\}}} \\ false & \text{, otherwise} \end{cases}$

- "active" is determined by the frame:

  $t(n) = \begin{cases} true & \text{, if } n = \boxed{C} \text{ or } n = \boxed{C} \\ false & \text{, otherwise} \end{cases}$

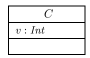
What If Things Are Missing?



- For instance, what about the box above?
- v has **no visibility**, **no initial value**, and (strictly speaking) **no properties**.

What If Things Are Missing?

- For instance, what about the box above?
- v has **no visibility**, **no initial value**, and (strictly speaking) **no properties**.



It depends.

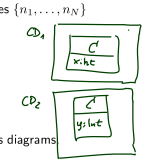
- What does the standard say? [OMG, 2007a, 121]

  "**Presentation Options.**
  The type, visibility, default, multiplicity, property string may be suppressed from being displayed, even if there are values in the model."

  - **Visibility**: There is no "no visibility" — an attribute **has** a visibility in the (extended) signature.
    Some (and we) assume **public** as default, but conventions may vary.
  - **Initial value**: some assume it **given by domain** (such as "leftmost value", but what is "leftmost" of $\mathbb{Z}$?).
    Some (and we) understand **non-deterministic initialisation**.
  - **Properties**: probably safe to assume $\emptyset$ if not given at all.

From Class Diagrams to Extended Signatures

- We view a **class diagram** $\mathcal{CD}$ as a graph with nodes $\{n_1, \dots, n_N\}$ (each "class rectangle" is a node).
  - $\mathscr{C}(\mathcal{CD}) := \bigcup_{i=1}^{N} \mathscr{C}(n_i)$ $= \{\mathscr{C}(n_i) \mid 1 \le i \le N\}$
  - $V(\mathcal{CD}) := \bigcup_{i=1}^{N} V(n_i)$
  - $atr(\mathcal{CD}) := \bigcup_{i=1}^{N} atr(n_i)$



- In a **UML model**, we can have **finitely many** class diagrams,

  $$\mathscr{C}\mathscr{D} = \{\mathcal{CD}_1, \dots, \mathcal{CD}_k\},$$

  which **induce** the following signature:

  $$\mathscr{S}(\mathscr{C}\mathscr{D}) = \left(\mathscr{T}, \bigcup_{i=1}^{k} \mathscr{C}(\mathcal{CD}_i), \bigcup_{i=1}^{k} V(\mathcal{CD}_i), \bigcup_{i=1}^{k} atr(\mathcal{CD}_i)\right).$$

  (Assuming $\mathscr{T}$ given. In "reality", we can introduce types in class diagrams, the class diagram then contributes to $\mathscr{T}$.)

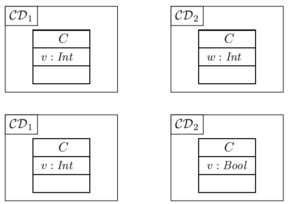
Is the Mapping a Function?

- Is $\mathscr{S}(\mathscr{CD})$ **well-defined**?

Two possible **sources for problems**:

(1) A **class** C may appear in **multiple** class **diagrams**:

(i) / (ii)



Simply **forbid** the case (ii) — easy syntactical check on diagram.

Is the Mapping a Function?

(2) An **attribute** v may appear in **multiple classes**:

Two approaches:

- Require **unique** attribute names.
  This requirement can easily be established (implicitly, behind the scenes) by viewing v as an abbreviation for

$$C::v \quad \text{or} \quad D::v$$

  depending on the context. ($C::v : Bool$ and $D::v : Int$ are unique.)
- Subtle, formalist's approach: observe that

$$\langle v : Bool, \dots \rangle \quad \text{and} \quad \langle v : Int, \dots \rangle$$

  are **different things** in V. But we don't follow that path...

Class Diagram Semantics

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$?

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$? **No.**

  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type τ, i.e. the set $\mathscr{D}(\tau)$, would be determined by the class diagram, and not free for choice.)

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$? **No.**

  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type τ, i.e. the set $\mathscr{D}(\tau)$, would be determined by the class diagram, and not free for choice.)
- What is the effect on $\Sigma_{\mathscr{S}}^{\mathscr{D}}$?

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$? **No.**
  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type τ, i.e. the set $\mathscr{D}(\tau)$, would be determined by the class diagram, and not free for choice.)
- What is the effect on $\Sigma_{\mathscr{S}}^{\mathscr{D}}$? **Little.**

  For now, we only **remove** abstract class instances, i.e.

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

  is now **only** called **system state** if and only if, for all $\langle C, S_C, 1, t\rangle \in \mathscr{C}$,

$$\mathrm{dom}(\sigma) \cap \mathscr{D}(C) = \emptyset.$$

  With $a = 0$ as default "abstractness", the earlier definitions apply directly.
  We'll revisit this when discussing inheritance.

What About The Rest?

- **Classes**:
  - **Active**: not represented in σ.
    Later: relevant for behaviour, i.e., how system states evolve over time.
  - **Stereotypes**: in a minute.
- **Attributes**:
  - **Initial value**: not represented in σ.
    Later: provides an initial value as effect of "creation action".
  - **Visibility**: not represented in σ.
    Later: viewed as additional **typing information** for well-formedness of system transformers; and with inheritance.
  - **Properties**: such as `readOnly`, `ordered`, `composite`
    (**Deprecated** in the standard.)
    - `readOnly` — **later** treated similar to visibility.
    - `ordered` — too fine for our representation.
    - `composite` — cf. lecture on associations.

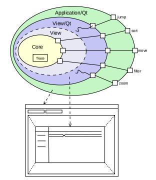
Stereotypes

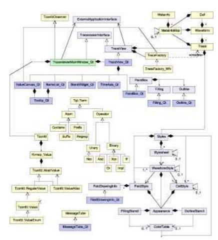
Stereotypes as Labels or Tags

- So, a class is
$$\langle C, S_C, a, t\rangle$$
  with a the abstractness flag, t activeness flag, and S_C a set of **stereotypes**.
- What are Stereotypes?
  - **Not** represented in system states.
  - **Not** contributing to typing rules.
    (cf. **later** lecture on type theory for UML)
- [Oestereich, 2006]:
  View stereotypes as (additional) "**labelling**" ("tags") or as "**grouping**".

  Useful for documentation and MDA.
  - **Documentation**: e.g. layers of an architecture.
    Sometimes, packages (cf. the standard) are sufficient and "right".
  - **Model Driven Architecture** (MDA): **later**.

Example: Stereotypes for Documentation

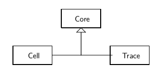




- Example: Timing Diagram Viewer
  [Schumann et al., 2008]
- Architecture of four layers:
  - core, data layer
  - abstract view layer
  - toolkit-specific view layer/widget
  - application using widget
- Stereotype "=" layer "=" colour

Stereotypes as Inheritance

- Another view (due to whom?): distinguish
  - **Technical Inheritance**
    If the **target platform**, such as the programming language for the implementation of the blueprint, is object-oriented, assume a 1-on-1 relation between inheritance in the model and on the target platform.
  - **Conceptual Inheritance**
    Only meaningful with a **common idea** of what stereotypes stand for. For instance, one could label each class with the team that is responsible for realising it. Or with licensing information (e.g., LGPL and proprietary).
    Or one could have labels understood by code generators (cf. lecture on MDSE).
- **Confusing**:
  - Inheritance is often referred to as the "is a"-relation.
    Sharing a stereotype also expresses "being something".
  - We can always (ab-)use UML-inheritance for the conceptual case, e.g.



Excursus: Type Theory (cf. Thiemann, 2008)

Type Theory

Recall: In lecture 03, we introduced OCL expressions with **types**, for instance:

$expr ::=$	w	$: \tau$	. . . logical variable w
	$\mid$ true $\mid$ false	$: Bool$	. . . constants
	$\mid 0 \mid -1 \mid 1 \mid \ldots$	$: Int$	. . . constants
	$\mid expr_1 + expr_2$	$: Int \times Int \to Int$	. . . operation
	$\mid$ size$(expr_1)$	$: Set(\tau) \to Int$	

Wanted: A procedure to tell **well-typed**, such as $(w : Bool)$

not w

from **not well-typed**, such as,

size(w).

Type Theory

Recall: In lecture 03, we introduced OCL expressions with **types**, for instance:

$expr ::=$	w	$: \tau$	. . . logical variable w
	$\mid$ true $\mid$ false	$: Bool$	. . . constants
	$\mid 0 \mid -1 \mid 1 \mid \ldots$	$: Int$	. . . constants
	$\mid expr_1 + expr_2$	$: Int \times Int \to Int$	. . . operation
	$\mid$ size$(expr_1)$	$: Set(\tau) \to Int$	

Wanted: A procedure to tell **well-typed**, such as $(w : Bool)$

not w

from **not well-typed**, such as,

size(w).

Approach: Derivation System, that is, a finite set of derivation rules.
We then say $expr$ **is well-typed** if and only if we can derive

$A, C \vdash expr : \tau$ (**read**: "expression $expr$ has type τ")

for some OCL type τ, i.e. $\tau \in T_B \cup T_{\mathscr{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathscr{C}}\}$, $C \in \mathscr{C}$.

A Type System for OCL

A Type System for OCL

We will give a finite set of **type rules** (a **type system**) of the form

$$(\text{"name"})\ \frac{\text{"premises"}}{\text{"conclusion"}}\ \text{"side condition"}$$

A Type System for OCL

We will give a finite set of **type rules** (a **type system**) of the form

$$(\text{"name"})\ \frac{\text{"premises"}}{\text{"conclusion"}}\ \text{"side condition"}$$

These rules will establish well-typedness statements (**type sentences**)
of three different "**qualities**":

(i) Universal well-typedness:

$$\vdash expr : \tau$$
$$\vdash 1 + 2 : Int$$

(ii) Well-typedness in a **type environment** A: (for logical variables)

$$A \vdash expr : \tau$$
$$self : \tau_C \vdash self.v : Int$$

(iii) Well-typedness in type environment A and **context** D: (for visibility)

$$A, D \vdash expr : \tau$$
$$self : \tau_C, C \vdash self\ .\ r\ .\ v : Int$$

Constants and Operations

- If $expr$ is a **boolean constant**, then $expr$ is of type $Bool$:

$$(BOOL) \quad \frac{}{\vdash B : Bool}, \quad B \in \{\mathit{true}, \mathit{false}\}$$

Constants and Operations

- If $expr$ is a **boolean constant**, then $expr$ is of type $Bool$:

$$(BOOL) \quad \frac{}{\vdash B : Bool}, \quad B \in \{\mathit{true}, \mathit{false}\}$$

- If $expr$ is an **integer constant**, then $expr$ is of type Int:

$$(INT) \quad \frac{}{\vdash N : Int}, \quad N \in \{0, 1, -1, \dots\}$$

Constants and Operations

- If $expr$ is a **boolean constant**, then $expr$ is of type $Bool$:

$$(BOOL) \quad \frac{}{\vdash B : Bool}, \quad B \in \{\mathit{true}, \mathit{false}\}$$

- If $expr$ is an **integer constant**, then $expr$ is of type Int:

$$(INT) \quad \frac{}{\vdash N : Int}, \quad N \in \{0, 1, -1, \dots\}$$

- If $expr$ is the application of **operation** $\omega : \tau_1 \times \dots \times \tau_n \to \tau$ to expressions $expr_1, \dots, expr_n$ which are of type $\tau_1, \dots, \tau_n$, then $expr$ is of type τ:

$$(Fun_0) \quad \frac{\vdash expr_1 : \tau_1 \ \dots \ \vdash expr_n : \tau_n}{\vdash \omega(expr_1, \dots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \dots \times \tau_n \to \tau, \\ n \geq 1, \ \omega \notin atr(\mathscr{C})\end{array}$$

(Note: this rule also covers '$=_\tau$', 'isEmpty', and 'size'.)

Constants and Operations Example

$$(BOOL) \quad \frac{}{\vdash B : Bool}, \quad B \in \{\mathit{true}, \mathit{false}\}$$

$$(INT) \quad \frac{}{\vdash N : Int}, \quad N \in \{0, 1, -1, \dots\}$$

$$(Fun_0) \quad \frac{\vdash expr_1 : \tau_1 \ \dots \ \vdash expr_n : \tau_n}{\vdash \omega(expr_1, \dots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \dots \times \tau_n \to \tau, \\ n \geq 1, \ \omega \notin atr(\mathscr{C})\end{array}$$

Example:

- not *true*

- *true* $+ 3$

Type Environment

- **Problem**: Whether

$$w + 3$$

is well-typed or not depends on the type of logical variable $w \in W$.

Type Environment

- **Problem**: Whether

$$w + 3$$

is well-typed or not depends on the type of logical variable $w \in W$.

- **Approach: Type Environments**

> **Definition.** A type environment is a (possibly empty) finite sequence of type declarations.
> The set of type environments for a given set W of logical variables and types T is defined by the grammar
>
> $$A ::= \emptyset \mid A, w : \tau$$
>
> where $w \in W$, $\tau \in T$.

Clear: We use this definition for the set of OCL logical variables W and the types $T = T_B \cup T_{\mathscr{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathscr{C}}\}$.

Environment Introduction and Logical Variables

- If $expr$ is of type τ, then it is of type τ **in any** type environment:

$$(EnvIntro) \quad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

Environment Introduction and Logical Variables

- If $expr$ is of type τ, then it is of type τ **in any** type environment:

$$(EnvIntro) \quad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

- Care for logical variables in **sub-expressions** of operator application:

$$(Fun_1) \quad \frac{A \vdash expr_1 : \tau_1 \ \ldots \ A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \cdots \times \tau_n \to \tau,\\ n \geq 1,\ \omega \notin atr(\mathscr{C})\end{array}$$

Environment Introduction and Logical Variables

- If $expr$ is of type τ, then it is of type τ **in any** type environment:

$$(EnvIntro) \quad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

- Care for logical variables in **sub-expressions** of operator application:

$$(Fun_1) \quad \frac{A \vdash expr_1 : \tau_1 \ \ldots \ A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \cdots \times \tau_n \to \tau,\\ n \geq 1,\ \omega \notin atr(\mathscr{C})\end{array}$$

- If $expr$ is a **logical variable** such that $w : \tau$ occurs in A, then we say w is of type τ,

$$(Var) \quad \frac{w : \tau \in A}{A \vdash w : \tau}$$

Type Environment Example

$$(EnvIntro) \quad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

$$(Fun_1) \quad \frac{A \vdash expr_1 : \tau_1 \ \ldots \ A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \cdots \times \tau_n \to \tau,\\ n \geq 1,\ \omega \notin atr(\mathscr{C})\end{array}$$

$$(Var) \quad \frac{w : \tau \in A}{A \vdash w : \tau}$$

Example:

- $w + 3,\ A = w : Int$

All Instances and Attributes in Type Environment

- If $expr$ refers to **all instances** of class C, then it is of type $Set(\tau_C)$,

$$(AllInst) \quad \frac{}{\vdash \mathsf{allInstances}_C : Set(\tau_C)}$$

All Instances and Attributes in Type Environment

- If $expr$ refers to **all instances** of class C, then it is of type $Set(\tau_C)$,

$$(AllInst) \quad \frac{}{\vdash \mathsf{allInstances}_C : Set(\tau_C)}$$

- If $expr$ is an **attribute access** of an attribute of type τ for an object of C as denoted by $expr_1$, then the premise is that $expr_1$ is of type τ_C:

$$(Attr_0) \quad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}, \quad v : \tau \in atr(C),\ \tau \in \mathscr{T}$$

$$(Attr_0^{0,1}) \quad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_1(expr_1) : \tau_D}, \quad r_1 : D_{0,1} \in atr(C)$$

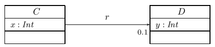
$$(Attr_0^{*}) \quad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \quad r_2 : D_{*} \in atr(C)$$

Attributes in Type Environment Example

$$(Attr_0)\quad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}, \qquad v : \tau \in atr(C),\ \tau \in \mathscr{T}$$

$$(Attr_0^{0,1})\quad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_1(expr_1) : \tau_D}, \qquad r_1 : D_{0,1} \in atr(C)$$

$$(Attr_0^{*})\quad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \qquad r_2 : D_* \in atr(C)$$



- $self : \tau_C \vdash self.x$
- $self : \tau_C \vdash self.r.x$
- $self : \tau_C \vdash self.r.y$
- $self : \tau_D \vdash self.x$

Iterate

- If $expr$ is an **iterate expression**, then
  - the iterator variable has to be type consistent with the base set, and
  - initial and update expressions have to be consistent with the result variable:

$$(Iter)\quad \frac{A \vdash expr_1 : Set(\tau_1) \quad A' \vdash expr_2 : \tau_2 \quad A' \vdash expr_3 : \tau_2}{A \vdash expr_1\text{->iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}$$

where $A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2)$.

Iterate Example

$$(AllInst)\quad \frac{}{\vdash \text{allInstances}_C : Set(\tau_C)} \qquad (Attr)\quad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}$$

$$(Iter)\quad \frac{A \vdash expr_1 : Set(\tau_1) \quad A' \vdash expr_2 : \tau_2 \quad A' \vdash expr_3 : \tau_2}{A \vdash expr_1\text{->iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}$$

where $A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2)$.

Example: $(\mathscr{T} = (\{Int\}, \{C\}, \{x : Int\}, \{C \mapsto \{x\}\})$

allInstances$_C$ −> iterate($self : C; v : Bool =$ true $\mid v \wedge self . x = 0$)

allInstances$_C$ −> forAll($self : C \mid self . x = 0$)

context $self : C$ inv : $self . x = 0$

context C inv : $x = 0$

First Recapitulation

- I **only** defined for well-typed expressions.
- **What can hinder** something, which looks like a well-typed OCL expression, from being a well-typed OCL expression...?

$$\mathscr{T} = (\{Int\}, \{C, D\}, \{x : Int, n : D_{0,1}\}, \{C \mapsto \{n\}, \{D \mapsto \{x\})$$

- Plain syntax error:

context C : *false*

- Subtle syntax error:

context C inv : $y = 0$

- Type error:

context $self : C$ inv : $self . n = self . n . x$

References

References

[Cabot and Clarisó, 2008] Cabot, J. and Clarisó, R. (2008). UML-OCL verification in practice. In Chaudron, M. R. V., editor, *MoDELS Workshops*, volume 5421 of *Lecture Notes in Computer Science*. Springer.

[Cengarle and Knapp, 2001] Cengarle, M. V. and Knapp, A. (2001). On the expressive power of pure OCL. Technical Report 0101, Institut für Informatik, Ludwig-Maximilians-Universität München.

[Cengarle and Knapp, 2002] Cengarle, M. V. and Knapp, A. (2002). Towards OCL/RT. In Eriksson, L.-H. and Lindsay, P. A., editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 390–409. Springer-Verlag.

[Flake and Müller, 2003] Flake, S. and Müller, W. (2003). Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling*, 2(3):164–186.

[Jackson, 2002] Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

[Oestereich, 2006] Oestereich, B. (2006). *Analyse und Design mit UML 2.1, 8. Auflage*. Oldenbourg, 8. edition.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.