

# *Software Design, Modelling and Analysis in UML*

## *Lecture 06: Type Systems and Visibility*

2012-11-13

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

Albert-Ludwigs-Universität Freiburg, Germany

- 06 - 2012-11-13 - main -

## Contents & Goals

### Last Lecture:

- Representing class diagrams as (extended) signatures — for the moment without associations (see Lectures 07 and 08).
- **And:** in Lecture 03, implicit assumption of well-typedness of OCL expressions.

### This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - Is this OCL expression well-typed or not? Why?
  - How/in what form did we define well-definedness?
  - What is visibility good for?
- **Content:**
  - Class diagram semantics.
  - Stereotypes – for documentation.
  - Recall: type theory/static type systems.
  - Well-typedness for OCL expression.
  - Visibility as a matter of well-typedness.

- 06 - 2012-11-13 - Prelim -

## Recall: From Class Boxes to Extended Signatures

### Extended Classes

From now on, we assume that each class  $C \in \mathcal{C}$  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  $S_{\mathcal{C}}$  to denote the set  $\bigcup_{C \in \mathcal{C}} S_C$  of stereotypes in  $\mathcal{S}$ .

(Alternatively, we could add a set  $St$  as 5-th component to  $\mathcal{S}$  to provides the stereotypes (names of stereotypes) to choose from. But: too unimportant to care.)

#### Convention:

- We write

$$\langle C, S_C, a, t \rangle \in \mathcal{C}$$

when we want to refer to all aspects of  $C$ .

- If the new aspects are irrelevant (for a given context), we simply write  $C \in \mathcal{C}$  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**

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

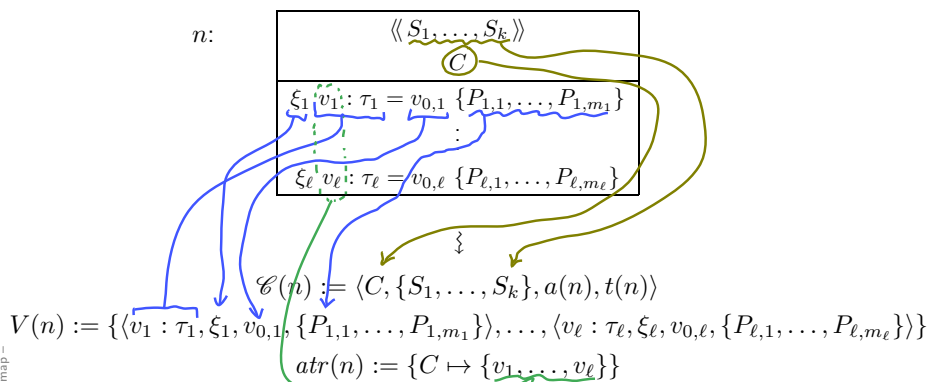
- an **initial value**  $expr_0$  given as a word from **language for initial values**, e.g. OCL expressions.  
(If using Java as **action language** (later) Java expressions would be fine.)
- a finite (possibly empty) set of **properties**  $P_v$ .  
We define  $P_\emptyset$  analogously to stereotypes.

### 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.

## From Class Boxes to Extended Signatures

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



where

- "abstract" is determined by the font:
- "active" is determined by the frame:

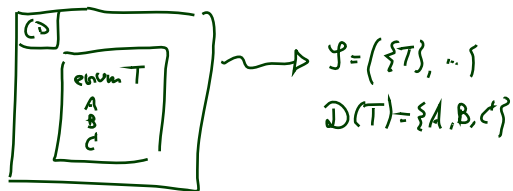
$$a(n) = \begin{cases} true & \text{, if } n = \boxed{C} \text{ or } n = \boxed{C \{A\}} \\ false & \text{, otherwise} \end{cases} \quad t(n) = \begin{cases} true & \text{, if } n = \boxed{C} \text{ or } n = \boxed{C} \\ false & \text{, otherwise} \end{cases}$$

## Class Diagram Semantics

### Semantics

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

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



## Semantics

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

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

$$\sigma : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(\mathcal{C}_*)))$$

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

$$\text{dom}(\sigma) \cap \mathcal{D}(C) = \emptyset.$$

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  $\sigma$ .  
**Later:** relevant for behaviour, i.e., how system states evolve over time.
  - **Stereotypes:** in a minute.
- **Attributes:**
  - **Initial value:** not represented in  $\sigma$ .  
**Later:** provides an initial value as effect of “creation action”.
  - **Visibility:** not represented in  $\sigma$ .  
**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. type theory for UML **later**)

- [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 already sufficient and "right".
- **Model Driven Architecture** (MDA): **later**.



## 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 $w$
<b>true</b>   <b>false</b>	$: Bool$	... constants
<b>0</b>   <b>-1</b>   <b>1</b>   ...	$: Int$	... constants
$expr_1 + expr_2$	$: Int \times Int \rightarrow Int$	... operation
<b>size</b> ( $expr_1$ )	$: Set(\tau) \rightarrow Int$	

**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  $\tau$ ")

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



## 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

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

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

(i) Universal well-typedness:

$$\begin{aligned} &\vdash \text{expr} : \tau \\ &\vdash 1 + 2 : \text{Int} \end{aligned}$$

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

$$\begin{aligned} &A \vdash \text{expr} : \tau \\ &\text{self} : \tau_C \vdash \text{self}.v : \text{Int} \end{aligned}$$

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

$$\begin{aligned} &A, B \vdash \text{expr} : \tau \\ &\text{self} : \tau_C, C \vdash \text{self}.r.v : \text{Int} \end{aligned}$$

## Constants and Operations

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

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

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

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

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

$$(Fun_0) \frac{\vdash expr_1 : \tau_1 \dots \vdash expr_n : \tau_n, \quad \omega : \tau_1 \times \dots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin atr(\mathcal{C})}{\vdash \omega(expr_1, \dots, expr_n) : \tau}$$

(Note: this rule also covers '=' <sub>$\tau$</sub> ', 'isEmpty', and 'size'.)

## Constants and Operations Example

(BOOL)	$\frac{}{\vdash B : Bool},$	$B \in \{true, false\}$
(INT)	$\frac{}{\vdash N : Int},$	$N \in \{0, 1, -1, \dots\}$
(Fun <sub>0</sub> )	$\frac{\vdash expr_1 : \tau_1 \dots \vdash expr_n : \tau_n,}{\vdash \omega(expr_1, \dots, expr_n) : \tau},$	$\omega : \tau_1 \times \dots \times \tau_n \rightarrow \tau,$ $n \geq 1, \omega \notin atr(\mathcal{C})$

### Example:

- not true

$$\frac{\vdash true : Bool}{\vdash not\ true : Bool}$$

- true + 3

$$\frac{\vdash true : Int \quad \vdash 3 : Int}{\vdash true + 3 : Int}$$

*for stack we cannot derive this from the rules*

*↳ true + 3 is not well-typed*

*isEmpty({1,2}):*  
*isEmpty({} (1,2))*

$$\frac{\frac{(Int) \frac{}{\vdash 1 : Int} \quad \frac{}{\vdash 2 : Int}}{(Fun_0) \frac{}{\vdash \{1, 2\} : Set(Int)}}}{(Fun_0) \frac{}{\vdash isEmpty(\{1, 2\}) : Bool}}$$

## 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  $T = T_B \cup T_{\mathcal{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathcal{C}}\}$ .

## Environment Introduction and Logical Variables

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

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

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

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

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

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

## Type Environment Example

(EnvIntro)	$\frac{\vdash \text{expr} : \tau}{A \vdash \text{expr} : \tau}$	
(Fun1)	$\frac{A \vdash \text{expr}_1 : \tau_1 \dots A \vdash \text{expr}_n : \tau_n}{A \vdash \omega(\text{expr}_1, \dots, \text{expr}_n) : \tau}, \quad \omega : \tau_1 \times \dots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{C})$	
(Var)	$\frac{w : \tau \in A}{A \vdash w : \tau}$	

### Example:

- $w + 3, A = w : \text{Int}$

$$\begin{array}{c}
 \text{(Var)} \quad \frac{w : \text{Int} \in A \quad w : \text{Int}}{w : \text{Int} \vdash w : \text{Int}} \quad \frac{\vdash 3 : \text{Int}}{\vdash 3 : \text{Int}} \text{(Int)} \\
 \frac{w : \text{Int} \vdash w : \text{Int} \quad w : \text{Int} \vdash 3 : \text{Int}}{w : \text{Int} \vdash w + 3 : \text{Int}} \text{(Fun1)} \\
 \frac{}{w : \text{Int} \vdash w + 3 : \text{Int}} \text{(EnvIntro)} \\
 \text{A}
 \end{array}$$

$\hookrightarrow w+3$  well-typed in type environment A

## All Instances and Attributes in Type Environment

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

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

- If  $\text{expr}$  is an **attribute access** of an attribute of type  $\tau$  for an object of  $C$  as denoted by  $\text{expr}_1$ , then the premise is that  $\text{expr}_1$  is of type  $\tau_C$ :

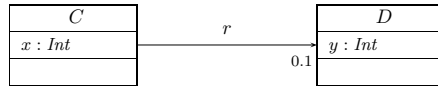
$$(\text{Attr}_0) \quad \frac{A \vdash \text{expr}_1 : \tau_C}{A \vdash v(\text{expr}_1) : \tau}, \quad v : \tau \in \text{atr}(C), \tau \in \mathcal{T}$$

$$(\text{Attr}_{0,1}^{0,1}) \quad \frac{A \vdash \text{expr}_1 : \tau_C}{A \vdash r_1(\text{expr}_1) : \tau_D}, \quad r_1 : D_{0,1} \in \text{atr}(C)$$

$$(\text{Attr}_0^*) \quad \frac{A \vdash \text{expr}_1 : \tau_C}{A \vdash r_2(\text{expr}_1) : \text{Set}(\tau_D)}, \quad r_2 : D_* \in \text{atr}(C)$$

## Attributes in Type Environment Example

$(Attr_0)$	$\frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}$	$v : \tau \in atr(C), \tau \in \mathcal{T}$
$(Attr_0^{0,1})$	$\frac{A \vdash expr_1 : \tau_C}{A \vdash r_1(expr_1) : \tau_D}$	$r_1 : D_{0,1} \in atr(C)$
$(Attr_0^*)$	$\frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}$	$r_2 : D_* \in atr(C)$



derivable,  
just check  
because  
 $y \in atr(D)$

not derivable  
from type  
environment

$self : \tau_C \vdash self : \tau_D$

$(Attr_0) \frac{self : \tau_C \vdash y(self) : Int}{self : \tau_C \vdash self : \tau_D}$   
↳ not well typed

- $self : \tau_C \vdash self.y : Int$
- $self : \tau_C \vdash self.x : Int$  well-typed by  $(Attr_0), (Var)$
- $self : \tau_C \vdash self.r : \tau_D$  well-typed by  $(Attr_1), (Var)$
- $self : \tau_C \vdash self.r.x : Int$  not well-typed,  $x \notin atr(D)$
- $self : \tau_C \vdash self.r.y : Int$  well-typed by

$\frac{self : \tau_D \in A \quad (Var)}{A \vdash self : \tau_D} \quad (Attr_0^{0,1})$   
 $\frac{A \vdash (self) : \tau_D}{A \vdash y(r(self)) : Int} \quad (Attr_0)$

- 06 - 2012-11-13 - Scedtyp -

## 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:

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

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

overide typing of  $w_1, w_2$  in  $A$   
( $w_1, w_2$  hide outer scope)

may use

- 06 - 2012-11-13 - Scedtyp -



## References

---

## References

- [Oestereich, 2006] Oestereich, B. (2006). *Analyse und Design mit UML 2.1*, 8. Auflage. Oldenbourg, 8. edition.
- [OMG, 2006] OMG (2006). Object Constraint Language, version 2.0. Technical Report formal/06-05-01.
- [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.
- [Schumann et al., 2008] Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). Traceviewer technical documentation, version 1.0. Technical report, Carl von Ossietzky Universität Oldenburg und OFFIS.