# Software Design, Modelling and Analysis in UML

## Lecture 07: A Type System for Visibility

2013-11-18

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

Albert-Ludwigs-Universität Freiburg, Germany



## Contents & Goals

**Last Lecture:**

- Representing class diagrams as (extended) signatures — for the moment without associations (see Lecture 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:**
  - Recall: type theory/static type systems.
  - Well-typedness for OCL expression.
  - Visibility as a matter of well-typedness.





## Recall: From Class Boxes to Extended Signatures





## 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.

**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:

$n$:

| $\langle\langle S_1, \dots, S_k \rangle\rangle$ <br> $C$ |
|---|
| $\xi_1\ v_1 : \tau_1 = v_{0,1}\ \{P_{1,1}, \dots, P_{1,m_1}\}$ <br> $\vdots$ <br> $\xi_\ell\ v_\ell : \tau_\ell = v_{0,\ell}\ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}$ |

$$\mathscr{C}(n) := \langle C, \{S_1, \dots, S_k\}, a(n), t(n) \rangle$$
$$V(n) := \{\langle v_1 : \tau_1, \xi_1, v_{0,1}, \{P_{1,1}, \dots, P_{1,m_1}\} \rangle, \dots, \langle v_\ell : \tau_\ell, \xi_\ell, v_{0,\ell}, \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\} \rangle\}$$
$$atr(n) := \{C \mapsto \{v_1, \dots, v_\ell\}\}$$

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

# Excursus: Type Theory (cf. Thiemann, 2008)





## Type Theory

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

$$
\begin{array}{lll}
expr ::= w & : \tau & \text{... logical variable } w \\
\quad \mid \textsf{true} \mid \textsf{false} & : Bool & \text{... constants} \\
\quad \mid 0 \mid -1 \mid 1 \mid \dots & : Int & \text{... constants} \\
\quad \mid expr_1 + expr_2 & : Int \times Int \to Int & \text{... operation} \\
\quad \mid \textsf{size}(expr_1) & : Set(\tau) \to Int & \\
\quad \mid \text{not } expr & : Bool \to Bool &
\end{array}
$$

**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. $\tau \in T_B \cup T_{\mathscr{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathscr{C}}\}$, $C \in \mathscr{C}$.





# 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

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

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** $B$: (for visibility)

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





## Constants and Operations

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

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

- 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 $\tau$:

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

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





## Constants and Operations Example

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

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

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

**Example**:

- not *true*

$$(Fun_0)\ \frac{(BOOL)\ \dfrac{}{\vdash true : Bool}}{\vdash not(true) : Bool}\ not : Bool \to Bool$$

- $true + 3$ got stuck – we cannot ① derive this from the rules

$$(Fun_0)\ \frac{\vdash true : Int \qquad \dfrac{}{\vdash 3 : Int}\ (INT)}{\vdash true + 3 : Int}\ + : Int \times Int \to Int$$

② ⇒ true + 3 is <u>not</u> well-typed

## 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 $\tau$, 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:

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

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

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



## 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**:

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

$$\dfrac{(Var)\,\dfrac{w : Int \in A}{A \vdash w : Int} \qquad (EnvIntro)\,\dfrac{(INT)\,\dfrac{}{\vdash 3 : Int}}{A \vdash 3 : Int}}{\underbrace{w : Int}_{=:A} \vdash w + 3 : Int}\,(Fun_1) \qquad + : Int \times Int \to Int$$

$+(w, 3)$ ← prefix normal form

$\omega$ $expr_1$ $expr_2$

⇒ $w+3$ is well-typed under $A$



## 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 $\tau$ for an object of $C$ as denoted by $expr_1$, then the premise is that $expr_1$ is of type $\tau_C$:

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

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

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



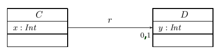
## Attributes in Type Environment Example

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

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

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

derivable but not useful




$V = \{x : Int,\ r : D_{0,1},\ y : Int\}$

$atr(C) = \{x, r\}$

$$\frac{self : \tau_C \vdash self : \tau_C \ (\tau_D)}{self : \tau_C \vdash self.y : Int}$$

needed but not derivable

⇒ get stuck

⇒ 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 $(Attr_0^{0,1})$, $(Var)$
- $self : \tau_C \vdash self.r.x : Int$ not well-typed, get stuck after applying $(Attr_0^{0,1})$
- $self : \tau_C \vdash self.r.y : Int$ well-typed by $(Attr_0^{0,1})$, $(Attr_0)$, $(Var)$



## 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:

well-typedness of $expr_2$ depends on outer scope

… inner scope

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

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

override typing of $w_1$ and $w_2$ in $A$
("$w_1 : \tau_1$, $w_2 : \tau_2$ hide outer scope")

outer scope

$allInst_C \to iterate(i\ \ldots \mid$

$i.r \to iterate(i\ \ldots \mid \ldots))$

inner scope

## Iterate Example

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

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

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

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

$$\frac{\frac{}{A \vdash \mathsf{allInstances}_C : Set(\tau_C)}(AllInst) \quad \frac{\frac{}{true : Bool}(Bool)}{A \vdash true : Bool}(Env\ mto) \quad \frac{\frac{r : Bool \in A'}{A' \vdash r : Bool} \quad \frac{\frac{\frac{self : \tau_C \in A'}{A' \vdash self : \tau_C}}{A' \vdash self(x) : Int} \quad \frac{\frac{}{\vdash 0 : Int}}{A' \vdash 0 : Int}}{A' \vdash =(self(x), 0) : Bool}}{\underbrace{r : Bool, self : \tau_C}_{=: A'} \vdash and(r, =(self(x), 0))}}{A \vdash \mathsf{allInstances}_C \to iterate(self : C; r : Bool = true \mid \underbrace{=(self(x), 0)}_{and(r, \quad)})}$$

$A \vdash$ context $C$ inv : $x = 0$

↳ well-typed



## 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...?

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

- Plain syntax error: "inv" missing
  context $C$ : *false*
- Subtle syntax error (depends on signature) not in $\mathscr{S}$
  context $C$ inv : $y = 0$
- Type error: $:D_{0,1}$ $:Int$
  context $self : C$ inv : $self\ .\ n = self\ .\ n\ .\ x$



## Casting in the Type System



## One Possible Extension: Implicit Casts

- We **may wish** to have

$$\vdash 1 \text{ and } \mathit{false} : Bool \qquad (*)$$

  **In other words**: We may wish that the type system allows to use $0, 1 : Int$ instead of *true* and *false* without breaking well-typedness.

- Then just have a rule:

$$(Cast)\quad \frac{A \vdash expr : Int}{A \vdash expr : Bool}$$

- With (Cast) (and (Int), and (Bool), and ($\text{Fun}_0$)), we can derive the sentence $(*)$, thus conclude well-typedness.

- **But**: that's only half of the story — the definition of the interpretation function $I$ that we have is not prepared, it doesn't tell us what $(*)$ means...



## Implicit Casts Cont'd

**So, why isn't there an interpretation for** (1 and *false*)?

- First of all, we have (syntax)

$$expr_1 \text{ and } expr_2 : Bool \times Bool \to Bool$$

- Thus,

$$I(\text{and}) : I(Bool) \times I(Bool) \to I(Bool)$$

  where $I(Bool) = \{\mathit{true}, \mathit{false}\} \cup \{\bot_{Bool}\}$.

- By definition,

$$I[\![1 \text{ and } \mathit{false}]\!](\sigma, \beta) = I(\text{and})(\quad I[\![1]\!](\sigma, \beta), \quad I[\![\mathit{false}]\!](\sigma, \beta) \quad),$$

  and **there we're stuck**.



## Implicit Casts: Quickfix

- Explicitly define

$$I[\![\text{and}(expr_1, expr_2)]\!](\sigma, \beta) := \begin{cases} b_1 \wedge b_2 & \text{, if } b_1 \neq \bot_{Bool} \neq b_2 \\ \bot_{Bool} & \text{, otherwise} \end{cases}$$

  where

  - $b_1 := toBool(I[\![expr_1]\!](\sigma, \beta))$,
  - $b_2 := toBool(I[\![expr_2]\!](\sigma, \beta))$,

  and where

$$toBool : I(Int) \cup I(Bool) \to I(Bool)$$

$$x \mapsto \begin{cases} \mathit{true} & \text{, if} \\ \mathit{false} & \text{, if} \\ \bot_{Bool} & \text{, otherwise} \end{cases}$$

## Bottomline

- There are **wishes** for the type-system which require changes in both, the definition of $I$ **and** the type system.
  In most cases not difficult, but tedious.
- **Note**: the extension is still a basic type system.
- **Note**: OCL has a far more elaborate type system which in particular addresses the relation between $Bool$ and $Int$ (cf. [OMG, 2006]).



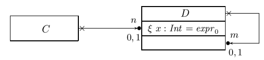
## Visibility in the Type System



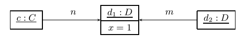
## Visibility — The Intuition

$$\mathscr{S} = (\{Int\}, \{C, D\}, \{n : D_{0,1}, m : D_{0,1}, \langle x : Int, \xi, expr_0, \emptyset\rangle\}, \{C \mapsto \{n\}, D \mapsto \{x, m\}\})$$

Let's study an **Example**:




and




Assume $w_1 : \tau_C$ and $w_2 : \tau_D$ are logical variables. **Which** of the following **syntactically correct** (?) OCL expressions **shall** we consider to be **well-typed**?

| $\xi$ of $x$: | public | private | protected | package |
|---|---|---|---|---|
| $w_1 . n . x = 0$ | ✔ | ✔ – | later | not |
| | ✘ | ✘ ⅢⅠ | | |
| | ? | ? rest | | |
| $w_2 . m . x = 0$ | ✔ | ✔ ⅢⅠ | later | not |
| $x(m(w_2)) = 0$ | ✘ | ✘ Ⅲ | | |
| | ? | ? rest | | |

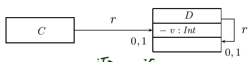
privateness is by class, not by object



## Context

$\mathscr{S} = (\{Int\}, \{C, D\}, \{r : D_{0,1}, v : Int\}, \{C \mapsto \{r\}, D \mapsto \{r, v\}\})$

- **Example**: A problem?




$$self : \tau_D \vdash self . r . v > 0 \quad ✓$$

$$self : \tau_C \not\vdash self . r . v > 0 \quad ✗$$

$:\tau_D$ $:\tau_D$ $:\tau_C$

- That is, whether an expression involving attributes with visibility is well-typed **depends** on the class of objects for which it is evaluated.
- **Therefore**: well-typedness in type environment $A$ and **context** $B \in \mathscr{C}$:

$$A, B \vdash expr : \tau$$

- In particular: prepare to treat "protected" later (when doing inheritance).



## Attribute Access in Context

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

$$(Context\boxed{Intro})\ \text{Drop} \qquad \frac{\overbrace{A}^{B,} \vdash expr : \tau}{A, \cancel{B} \vdash expr : \tau}$$

- **Accessing attribute** $v$ of a $C$-object via logical variable $w$ is well-typed if
  - ~~$v$ is public, or~~ $w$ is of type $\tau_B$

$$(Attr_1) \quad \frac{A \vdash w : \tau_B}{A, B \vdash v(w) : \tau}, \quad \langle v : \tau, \xi, expr_0, P_{\mathscr{C}}\rangle \in atr(B)$$

- **Accessing attribute** $v$ of a $C$-object of via expression $expr_1$ is well-typed **in context** $B$ if
  - $v$ is public, **or** $expr_1$ denotes an object of class $B$:

$$(Attr_2) \quad \frac{A, B \vdash expr_1 : \tau_C}{A, B \vdash v(expr_1) : \tau}, \quad \langle v : \tau, \xi, expr_0, P_{\mathscr{C}}\rangle \in atr(C), \ \xi = +, \text{ or } C = B$$

- Accessing $C_{0,1}$- or $C_*$-typed attributes: similar.



## Context in Operator Application

- Operator Application:

$$(Fun_2) \quad \frac{A, B \vdash expr_1 : \tau_1 \ \ldots \ A, B \vdash expr_n : \tau_n}{A, B \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \ n \geq 1, \ \omega \notin atr(\mathscr{C})$$

- Iterate:

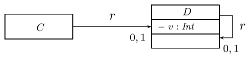
$$(Iter_1) \quad \frac{A, B \vdash expr_1 : Set(\tau_1) \quad A', B \vdash expr_2 : \tau_2 \quad A', B \vdash expr_3 : \tau_2}{A, B \vdash expr_1\text{->iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}$$

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

## Attribute Access in Context Example

$(Context\cancel{Intro})$ Drop $\dfrac{A{,B} \vdash expr : \tau}{A, \cancel{B} \vdash expr : \tau}$

$(Attr_1)$ $\dfrac{A, B \vdash expr_1 : \tau_C}{A, B \vdash v(expr_1) : \tau}$, $\langle v : \tau, \xi, expr_0, P_{\mathscr{C}} \rangle \in atr(C)$, $\xi = +$, or $\xi = -$ and $C = B$




**Example**:

$self : \tau_C \quad \vdash self . r . v > 0$



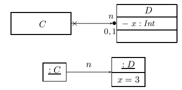
## The Semantics of Visibility

- **Observation**:
  - Whether an expression **does** or **does not** respect visibility is a matter of well-typedness **only**.
  - We only evaluate (= apply $I$ to) **well-typed** expressions.

$\rightarrow$ We **need not** adjust the interpretation function $I$ to support visibility.



## What is Visibility Good For?




- Visibility is a property of attributes — is it useful to consider it in OCL?
- In other words: given the picture above, **is it useful** to state the following invariant (even though $x$ is private in $D$)

context $C$ inv : $n.x > 0$ ?

- **It depends.** (cf. [OMG, 2006], Sect. 12 and 9.2.2)
  - **Constraints and pre/post conditions**:
    - Visibility is **sometimes** not taken into account. To state "global" requirements, it may be adequate to have a "global view", be able to look into all objects.
    - But: visibility supports "narrow interfaces", "information hiding", and similar good design practices. To be more robust against changes, try to state requirements only in the terms which are visible to a class.

    **Rule-of-thumb**: if attributes are important to state requirements on design models, leave them public or provide get-methods (later).
  - **Guards** and **operation bodies**:
    If in doubt, **yes** (= do take visibility into account).
    Any so-called **action language** typically takes visibility into account.



# Recapitulation



## Recapitulation

Class Diagrams $\mathscr{CD}$
$\downarrow$ **induces**
extended (!) signature $\mathscr{S}(\mathscr{CD})$
$\downarrow$ **gives rise to**
Basic Type System

- We extended the type system for
  - **casts** (requires change of $I$) and $\triangleleft$ see earlier slides
  - **visibility** (no change of $I$).
- **Later**: **navigability** of associations.

**Good**: well-typedness is decidable for these type-systems. That is, we can have automatic tools that check, whether OCL expressions in a model are well-typed.



# References

# References

[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.