Software Design, Modelling and Analysis in UML

Lecture 6: Class Diagrams I

2015-11-12

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

Albert-Ludwigs-Universität Freiburg, Germany

Course Map

UML

Model

$$\begin{tikzcd}
 & \mathcal{CD},\,\mathcal{SM} \arrow[dl, squiggly, "!"'] \arrow[dr, squiggly] & & \varphi \in \mathsf{OCL} \arrow[d, squiggly, leftrightarrow, "\checkmark"] & & \mathcal{CD},\,\mathcal{SD} \arrow[d, squiggly] \\
\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr) \arrow[d, squiggly, "\checkmark"'] \arrow[d, squiggly, shift left] & & SM \arrow[dll, squiggly] & expr & & \mathscr{S},\, SD \arrow[d, squiggly] \arrow[d, squiggly, shift left] \arrow[d, squiggly, shift left=2] \\
M = (\Sigma_{\mathscr{S}}^{\mathscr{D}}, A_{\mathscr{S}}, \to_{SM}) \arrow[ddr, squiggly] \arrow[ddr, squiggly, shift left, "\checkmark"] & & & & & B = (Q_{SD}, q_0, A_{\mathscr{S}}, \to_{SD}, F_{SD}) \\
 & & & & & \\
 & \pi = (\sigma_0, \varepsilon_0) \arrow[r, "{(cons_0, Snd_0)}", "u_0"'] \arrow[uuurr, squiggly, "\checkmark"] & (\sigma_1, \varepsilon_1) \cdots \arrow[rrr, squiggly, leftrightarrow] & & & w_\pi = ((\sigma_i, cons_i, Snd_i))_{i \in \mathbb{N}} \arrow[uu, squiggly] \\
 & & & G = (N, E, f) \arrow[ull, squiggly, "\checkmark"'] \arrow[d, squiggly, leftrightarrow, "\checkmark"'] & & \\
 & & & \mathcal{OD} & &
\end{tikzcd}$$

Instances

Mathematics

UML

Contents & Goals

Last Lecture:

- Object Diagrams
  - partial vs. complete; for analysis; for documentation. . .

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:**
  - Study UML syntax.
  - Prepare (extend) definition of signature.
  - Map class diagram to (extended) signature.
  - Stereotypes.

UML Class Diagrams: Stocktaking

Recall: Signature vs. Class Diagram

Basic Object System Signature Another Example

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

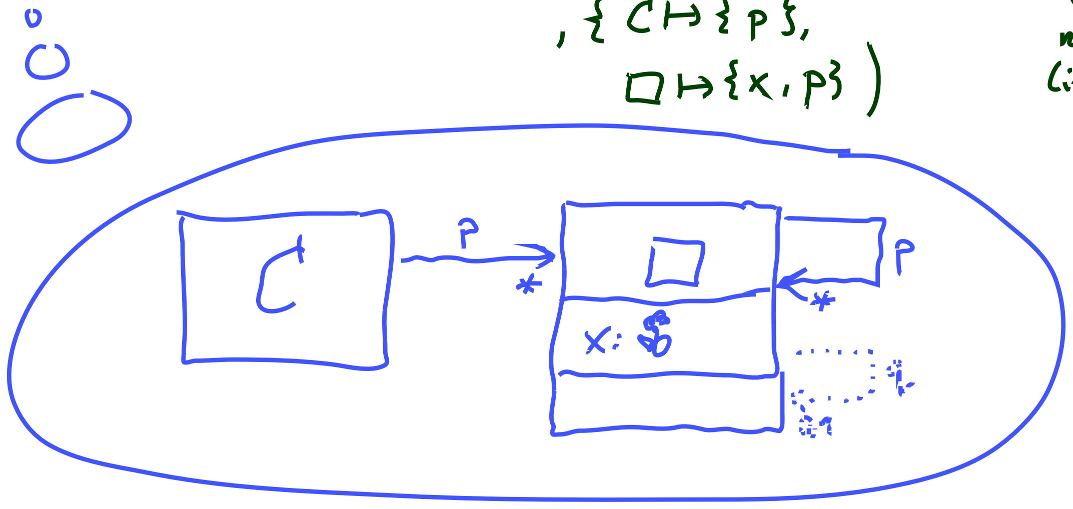
- (basic) types $\mathscr{T}$ and classes $\mathscr{C}$ (both finite),
- typed attributes V, τ from $\mathscr{T}$, or $C_{0,1}$ or C_*, for some $C \in \mathscr{C}$,
- $atr : \mathscr{C} \to 2^V$ mapping classes to attributes.

Example:

no, Int $\notin \mathscr{T}$ of $\mathscr{S}_1$

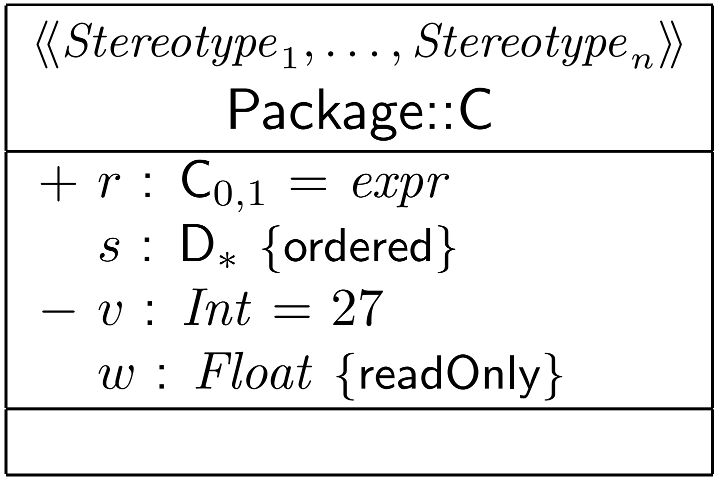
$$\mathscr{S}_1 = (\{\$, MyType\}, \{C, \square\}, \{\cancel{x : Int}\ x : \$, p : \square_*, q : \square_{0,1}\}, \{C \mapsto \{p\}, \square \mapsto \{x, p\}\})$$

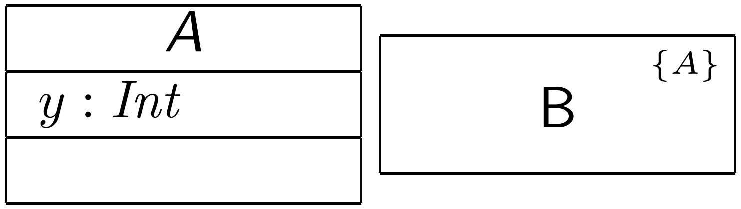
need not be used (if not used, may be omitted)

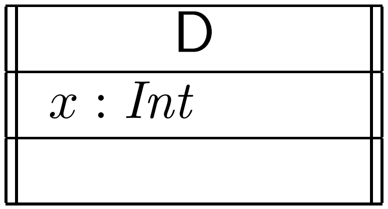


looks like a class diagram...

That'd Be Too Simple



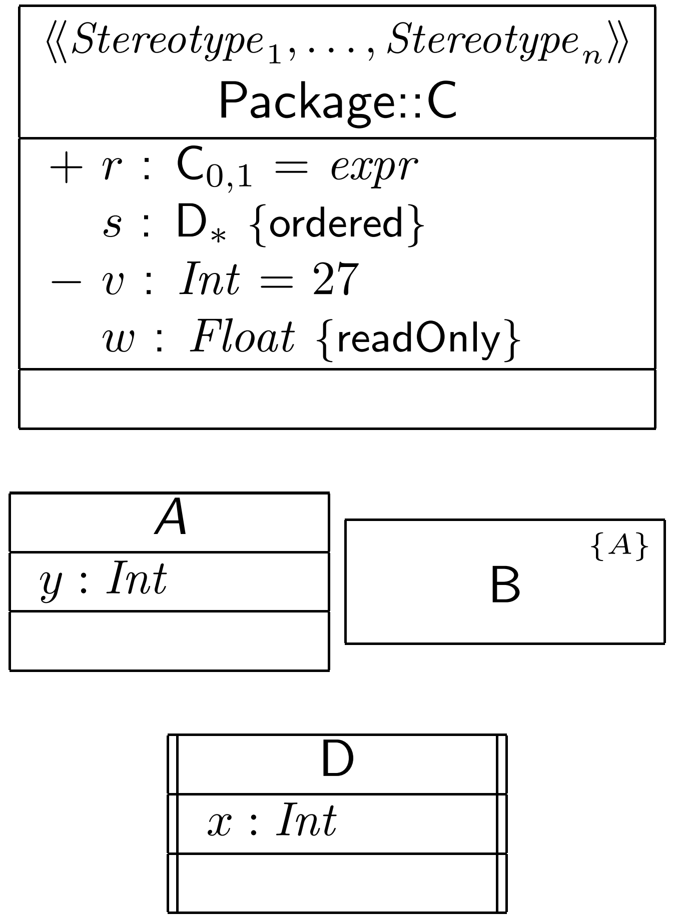




What Do We Want / Have to Cover?

A **class**

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



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

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes
- V is a finite set of attributes
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes $\langle C, S_C, a, t \rangle$ where
- V is a finite set of attributes
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes $\langle C, S_C, a, t \rangle$ where
  - S_C is a finite (possibly empty) set of **stereotypes**,
  - $a \in \mathbb{B}$ is a boolean flag indicating whether C is **abstract**,
  - $t \in \mathbb{B}$ is a boolean flag indicating whether C is **active**,
- V is a finite set of attributes
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes $\langle C, S_C, a, t\rangle$ where
  - S_C is a finite (possibly empty) set of **stereotypes**,
  - $a \in \mathbb{B}$ is a boolean flag indicating whether C is **abstract**,
  - $t \in \mathbb{B}$ is a boolean flag indicating whether C is **active**,
- V is a finite set of attributes $\langle v : T, \xi, expr_0, P_v\rangle$ where
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes $\langle C, S_C, a, t\rangle$ where
  - S_C is a finite (possibly empty) set of **stereotypes**,
  - $a \in \mathbb{B}$ is a boolean flag indicating whether C is **abstract**,
  - $t \in \mathbb{B}$ is a boolean flag indicating whether C is **active**,
- V is a finite set of attributes $\langle v : T, \xi, expr_0, P_v\rangle$ where
  - T is a type from $\mathscr{T}$, or $C_{0,1}, C_*$ for some $C \in \mathscr{C}$,
  - $\xi \in \{\underbrace{\text{public}}_{:=+}, \underbrace{\text{private}}_{:=-}, \underbrace{\text{protected}}_{:=\#}, \underbrace{\text{package}}_{:=\sim}\}$ is the **visibility**,
  - an **initial value expression** $expr_0$ given as a word from a **language for initial value expressions**, e.g. OCL, or C++ in the Rhapsody tool,
  - a finite (possibly empty) set of **properties** P_v.
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

Extended Signature

Definition. An (Extended) Object System Signature is a quadruple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr)$ where

- $\mathscr{T}$ is a set of (basic) types,
- $\mathscr{C}$ is a finite set of classes $\langle C, S_C, a, t\rangle$ where
  - S_C is a finite (possibly empty) set of **stereotypes**,
  - $a \in \mathbb{B}$ is a boolean flag indicating whether C is **abstract**,
  - $t \in \mathbb{B}$ is a boolean flag indicating whether C is **active**,
- V is a finite set of attributes $\langle v : T, \xi, expr_0, P_v\rangle$ where
  - T is a type from $\mathscr{T}$, or $C_{0,1}, C_*$ for some $C \in \mathscr{C}$,
  - $\xi \in \{\underbrace{\text{public}}_{:=+}, \underbrace{\text{private}}_{:=-}, \underbrace{\text{protected}}_{:=\#}, \underbrace{\text{package}}_{:=\sim}\}$ is the **visibility**,
  - an **initial value expression** $expr_0$ given as a word from a **language for initial value expressions**, e.g. OCL, or C++ in the Rhapsody tool,
  - a finite (possibly empty) set of **properties** P_v.
- $atr : \mathscr{C} \to 2^V$ maps each class to its set of attributes.

We use $S_{\mathscr{C}}$ to denote the set $\bigcup_{C \in \mathscr{C}} S_C$ of stereotypes in $\mathscr{S}$.

Conventions

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

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

Conventions

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

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

- **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 will remain mostly unchanged.
- **The other way round**: **most** of the newly added aspects **do not contribute** to the constitution of system states or object diagrams.

Mapping UML Class Diagrams to Extended Signatures

From Class Boxes to Extended Signatures

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

n:

$$\begin{array}{|c|}
\hline
\langle\!\langle\, S_1, \dots, S_k \,\rangle\!\rangle \\
C \\
\hline
\xi_1 \ v_1 : T_1 = expr_0^1 \ \{P_{1,1}, \dots, P_{1,m_1}\} \\
\vdots \\
\xi_\ell \ v_\ell : T_\ell = expr_0^\ell \ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\} \\
\hline
\end{array}$$

From Class Boxes to Extended Signatures

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

$$n: \quad \begin{array}{|c|}\hline \langle\!\langle\, S_1, \dots, S_k \,\rangle\!\rangle \\ C \\ \hline \xi_1 \ v_1 : T_1 = \mathit{expr}_0^1 \ \{P_{1,1}, \dots, P_{1,m_1}\} \\ \vdots \\ \xi_\ell \ v_\ell : T_\ell = \mathit{expr}_0^\ell \ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\} \\ \hline \end{array}$$

$$\rightsquigarrow$$

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

$$V(n) := \{\langle v_1 : T_1, \xi_1, \mathit{expr}_0^1, \{P_{1,1}, \dots, P_{1,m_1}\}\rangle, \dots, \langle v_\ell : T_\ell, \xi_\ell, \mathit{expr}_0^\ell, \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}\rangle\}$$

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

where

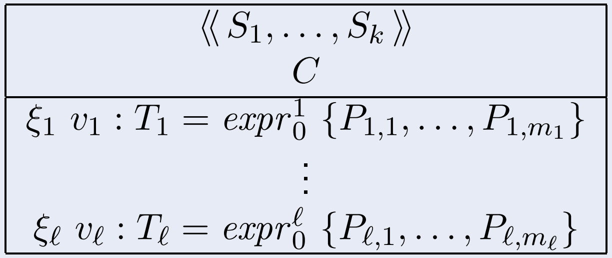
- "abstract" is determined by the font:

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

- "active" is determined by the frame:

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

Example



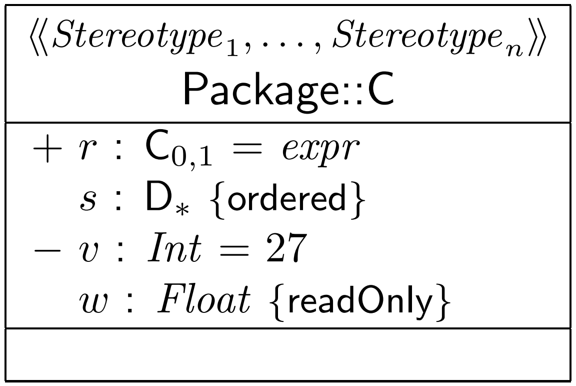
$\langle\langle\, S_1, \dots, S_k \,\rangle\rangle$
C
$\xi_1\ v_1 : T_1 = expr_0^1\ \{P_{1,1}, \dots, P_{1,m_1}\}$
$\vdots$
$\xi_\ell\ v_\ell : T_\ell = expr_0^\ell\ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}$

$$\rightsquigarrow$$

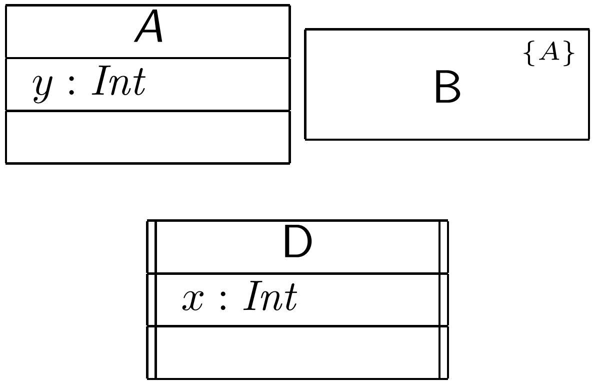
$$C(n) := \langle C, \{S_1, \dots, S_k\}, a(n), t(n) \rangle$$

$$V(n) := \{\langle v_1 : T_1, \xi_1, expr_0^1, \{P_{1,1}, \dots, P_{1,m_1}\}\rangle, \dots, \langle v_\ell : T_\ell, \xi_\ell, expr_0^\ell, \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}\rangle\}$$

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



$\langle\langle Stereotype_1, \dots, Stereotype_n \rangle\rangle$
Package::C
$+\ r : \mathsf{C}_{0,1} = expr$
$s : \mathsf{D}_* \ \{\mathsf{ordered}\}$
$-\ v : Int = 27$
$w : Float\ \{\mathsf{readOnly}\}$



What If Things Are Missing?

It depends.

- What does the standard say? (OMG, 2011a, 121)

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

What If Things Are Missing?

It depends.

- What does the standard say? (OMG, 2011a, 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.

What If Things Are Missing?

It depends.

- What does the standard say? (OMG, 2011a, 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** if not given.

What If Things Are Missing?

It depends.

- What does the standard say? (OMG, 2011a, 121)

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

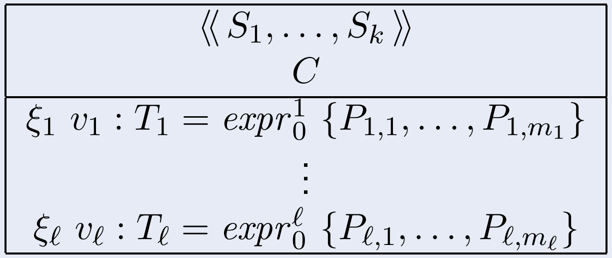
- **Visibility**: There is no "no visibility" — an attribute **has** a visibility in the (extended) signature.

  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** if not given.

- **Properties**: probably safe to assume $\emptyset$ if not given at all.

Example Cont'd



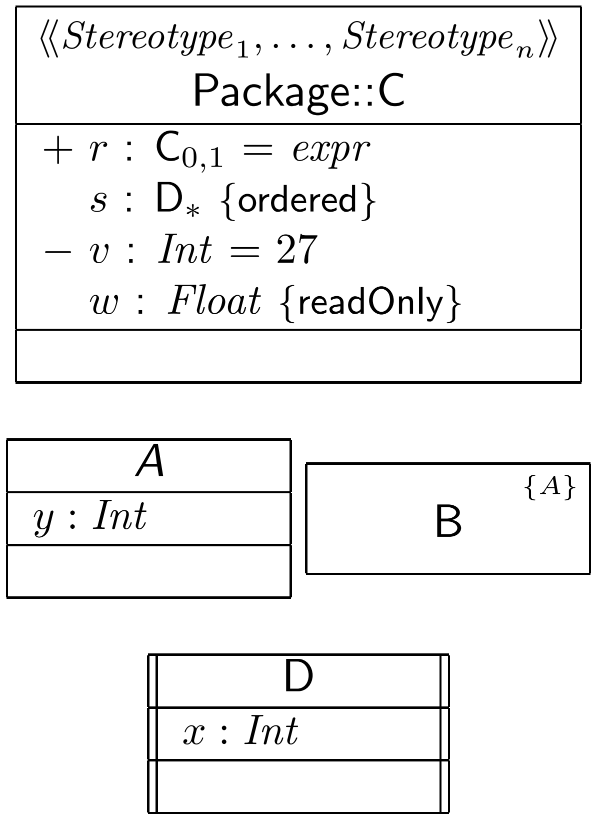
$\langle\!\langle S_1, \dots, S_k \rangle\!\rangle$
C
$\xi_1\ v_1 : T_1 = expr_0^1\ \{P_{1,1}, \dots, P_{1,m_1}\}$
$\vdots$
$\xi_\ell\ v_\ell : T_\ell = expr_0^\ell\ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}$

$$\rightsquigarrow$$

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

$$V(n) := \{\langle v_1 : T_1, \xi_1, expr_0^1, \{P_{1,1}, \dots, P_{1,m_1}\}\rangle, \dots, \langle v_\ell : T_\ell, \xi_\ell, expr_0^\ell, \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}\rangle\}$$

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



$\langle\!\langle Stereotype_1, \dots, Stereotype_n \rangle\!\rangle$
Package::C
$+\ r : \mathsf{C}_{0,1} = expr$
$s : \mathsf{D}_* \{\mathsf{ordered}\}$
$-\ v : Int = 27$
$w : Float \{\mathsf{readOnly}\}$
A
$y : Int$
B
$\{A\}$
D
$x : Int$

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).
  - $\mathscr{C}(\mathcal{CD}) := \{C(n_i) \mid 1 \leq i \leq N\}$
  - $V(\mathcal{CD}) := \bigcup_{i=1}^{N} V(n_i)$
  - $atr(\mathcal{CD}) := \bigcup_{i=1}^{N} atr(n_i)$

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).
  - $\mathscr{C}(\mathcal{CD}) := \{C(n_i) \mid 1 \leq i \leq N\}$
  - $V(\mathcal{CD}) := \bigcup_{i=1}^{N} V(n_i)$
  - $atr(\mathcal{CD}) := \bigcup_{i=1}^{N} atr(n_i)$

- 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" (i.e. in full UML), we can introduce types in class diagrams, the class diagram then contributes to $\mathscr{T}$. Example: enumeration types.)

Is the Mapping a Function?

Question: Is $\mathscr{S}(\mathscr{C}\mathscr{D})$ **well-defined**?

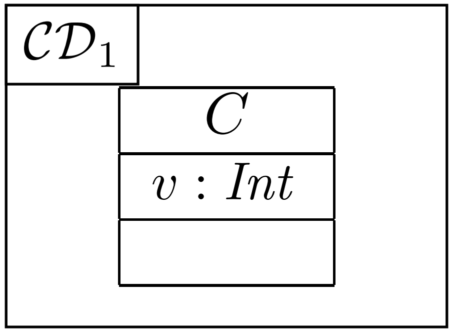
Is the Mapping a Function?

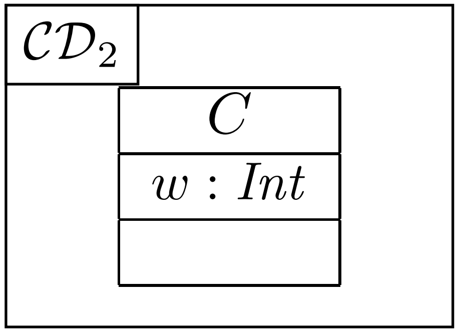
Question: Is $\mathscr{S}(\mathscr{CD})$ **well-defined**?

There are two possible **sources for problems**:

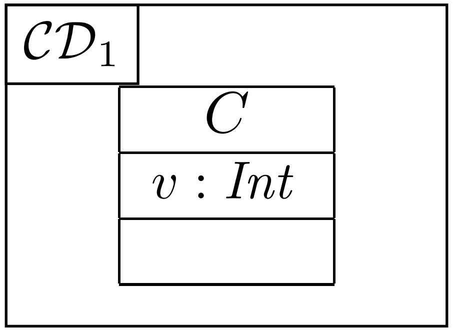
(1) A **class** C may appear in **multiple** class **diagrams**:

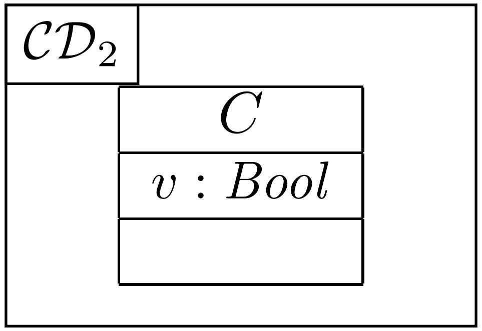
(i)





(ii)





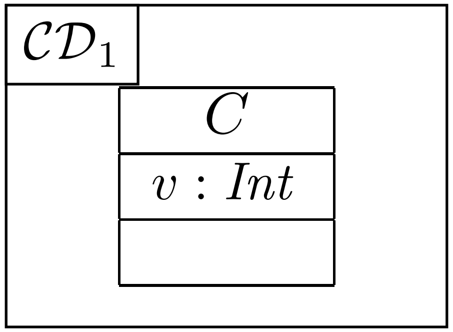
Is the Mapping a Function?

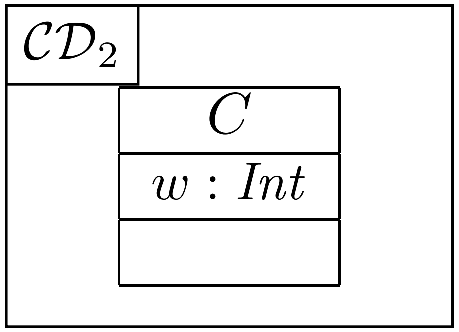
Question: Is $\mathscr{S}(\mathscr{CD})$ **well-defined**?

There are two possible **sources for problems**:

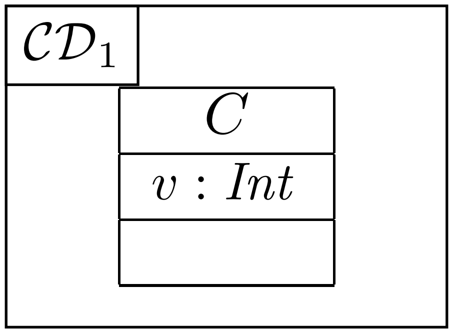
(1) A **class** C may appear in **multiple** class **diagrams**:

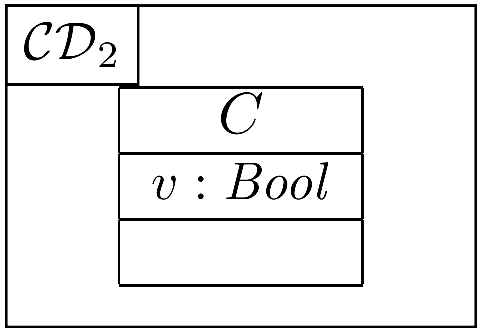
(i)





(ii)

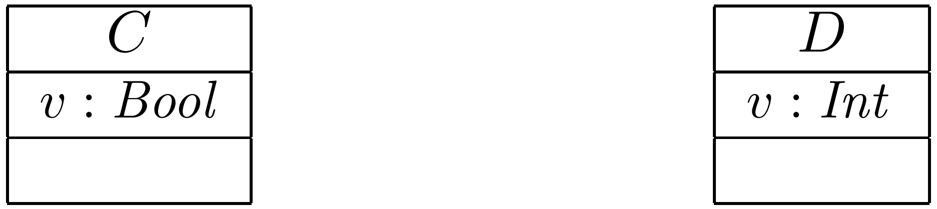




Simply **forbid** the case (ii) — easy syntactical check on diagram.

Is the Mapping a Function?

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



Two approaches:

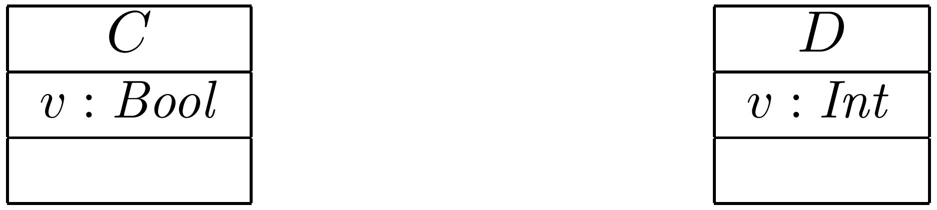
- Require **unique** attribute names.
  This requirement can easily be established (implicitly, behind the scenes) by viewing v as an abbreviation for

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

  depending on the context. ($C::v : Bool$ and $D::v : Int$ are then unique.)

Is the Mapping a Function?

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



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 then 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. We don't follow that path...

Class Diagram Semantics

Semantics

The semantics of a set of **class diagrams** $\mathscr{CD}$ is the induced **signature** $\mathscr{S}(\mathscr{CD})$.

Semantics

The semantics of a set of **class diagrams** $\mathscr{C}\mathscr{D}$ is the induced **signature** $\mathscr{S}(\mathscr{C}\mathscr{D})$.

The **signature** induces a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ (given a **structure** $\mathscr{D}$).

Semantics

The semantics of a set of **class diagrams** $\mathscr{CD}$ is the induced **signature** $\mathscr{S}(\mathscr{CD})$.

The **signature** induces a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ (given a **structure** $\mathscr{D}$).

- Do we need to redefine/extend $\mathscr{D}$?

Semantics

The semantics of a set of **class diagrams** $\mathscr{CD}$ is the induced **signature** $\mathscr{S}(\mathscr{CD})$.

The **signature** induces a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ (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 T, i.e. the set $\mathscr{D}(T)$, would be determined by the class diagram, and not free for choice.)

Semantics

The semantics of a set of **class diagrams** $\mathscr{CD}$ is the induced **signature** $\mathscr{S}(\mathscr{CD})$.

The **signature** induces a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ (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 T, i.e. the set $\mathscr{D}(T)$, 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** $\mathscr{C}\mathscr{D}$ is the induced **signature** $\mathscr{S}(\mathscr{C}\mathscr{D})$.

The **signature** induces a set of **system states** $\Sigma_{\mathscr{S}}^{\mathscr{D}}$ (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 T, i.e. the set $\mathscr{D}(T)$, 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**:
  - **Stereotypes**:

What About The Rest?

- **Classes**:
  - **Active**: not represented in σ.

    Later: relevant for behaviour, i.e., how system states evolve over time.
  - **Stereotypes**:

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.

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 expression**:
  - **Visibility**:
  - **Properties**:

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 expression**: not represented in σ.

    Later: provides an initial value as effect of "creation action".
  - **Visibility**:
  - **Properties**:

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 expression**: 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 actions; and with inheritance.
  - **Properties**:

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 expression**: 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 actions; and with inheritance.
  - **Properties**: such as `readOnly`, `ordered`, `composite` (**Deprecated** in the standard.)
    - `readOnly` — **later** treated similar to visibility.
    - `ordered` — not considered in our UML fragment ($\rightarrow$ sets vs. sequences).
    - `composite` — cf. lecture on associations.

Stereotypes

Stereotypes as Labels or Tags

- What are Stereotypes?
  - **Not** represented in system states.
  - **Not** contributing to typing rules / well-formedness.

Stereotypes as Labels or Tags

- What are Stereotypes?
  - **Not** represented in system states.
  - **Not** contributing to typing rules / well-formedness.

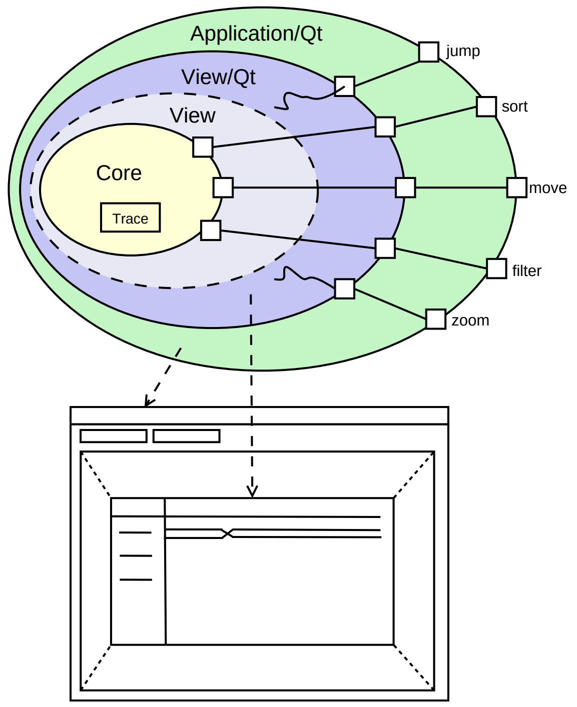
- Oestereich (2006):

  View stereotypes as (additional) "**labelling**" ("tags") or as "**grouping**".

- Useful for documentation and model-driven development, e.g. code-generation:
  - **Documentation**: e.g. layers of an architecture.

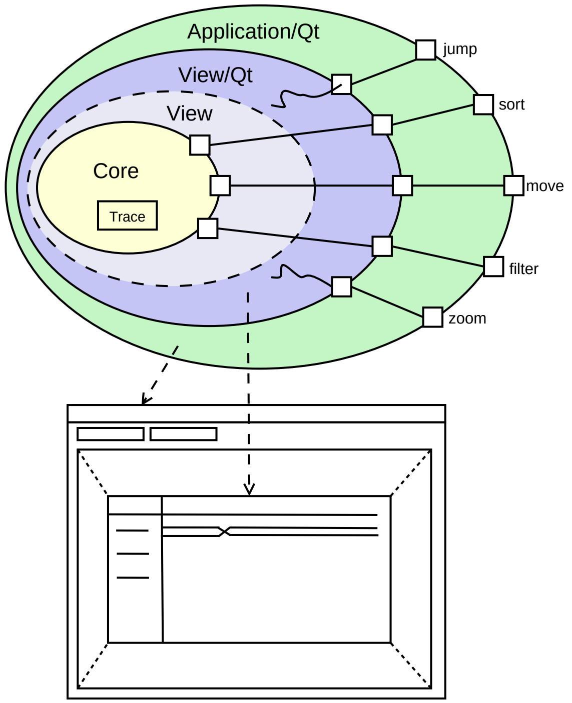
    Sometimes, packages (cf. OMG (2011a,b)) are sufficient and "right".
  - **Model Driven Architecture** (MDA): **later**.

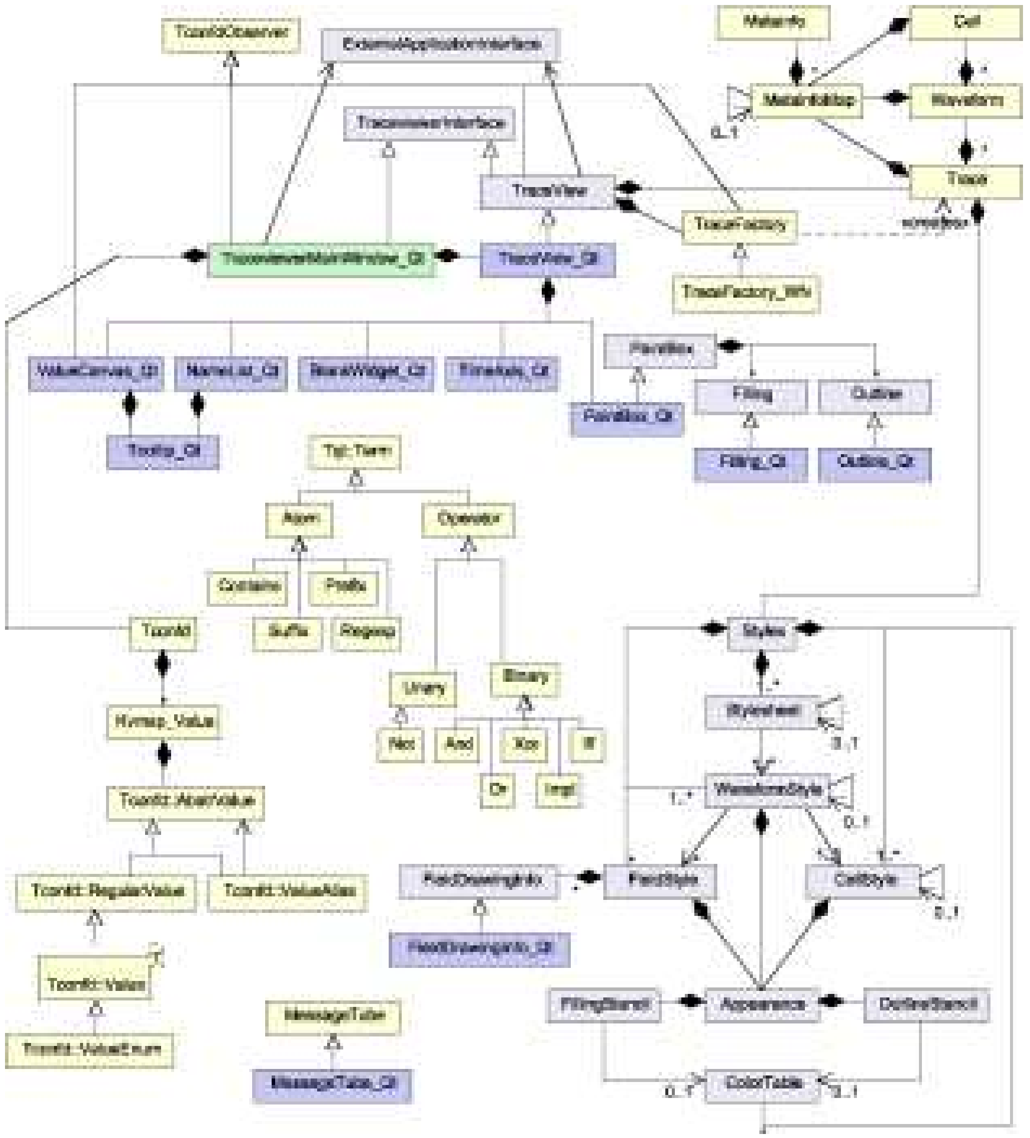
Example: Stereotypes for Documentation



- **Example**: Timing Diagram Viewer
  Schumann et al. (2008)
- Architecture has four layers:
  - core, data layer
  - abstract view layer
  - toolkit-specific view layer/widget
  - application using widget

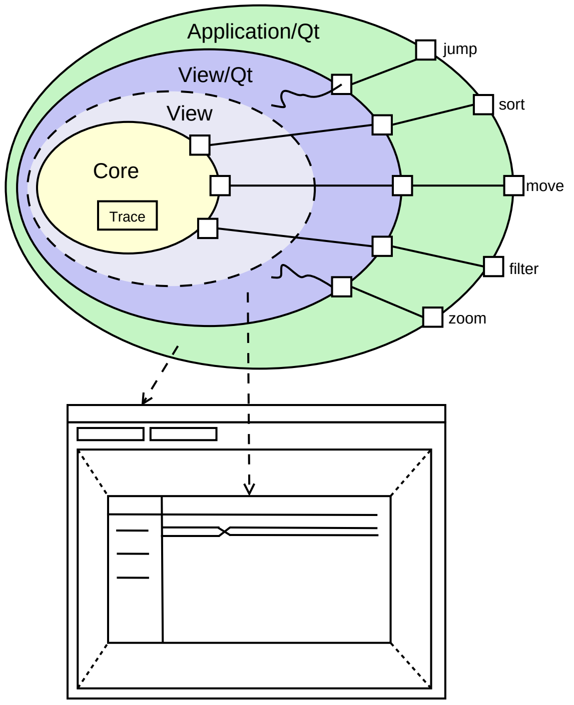
Example: Stereotypes for Documentation

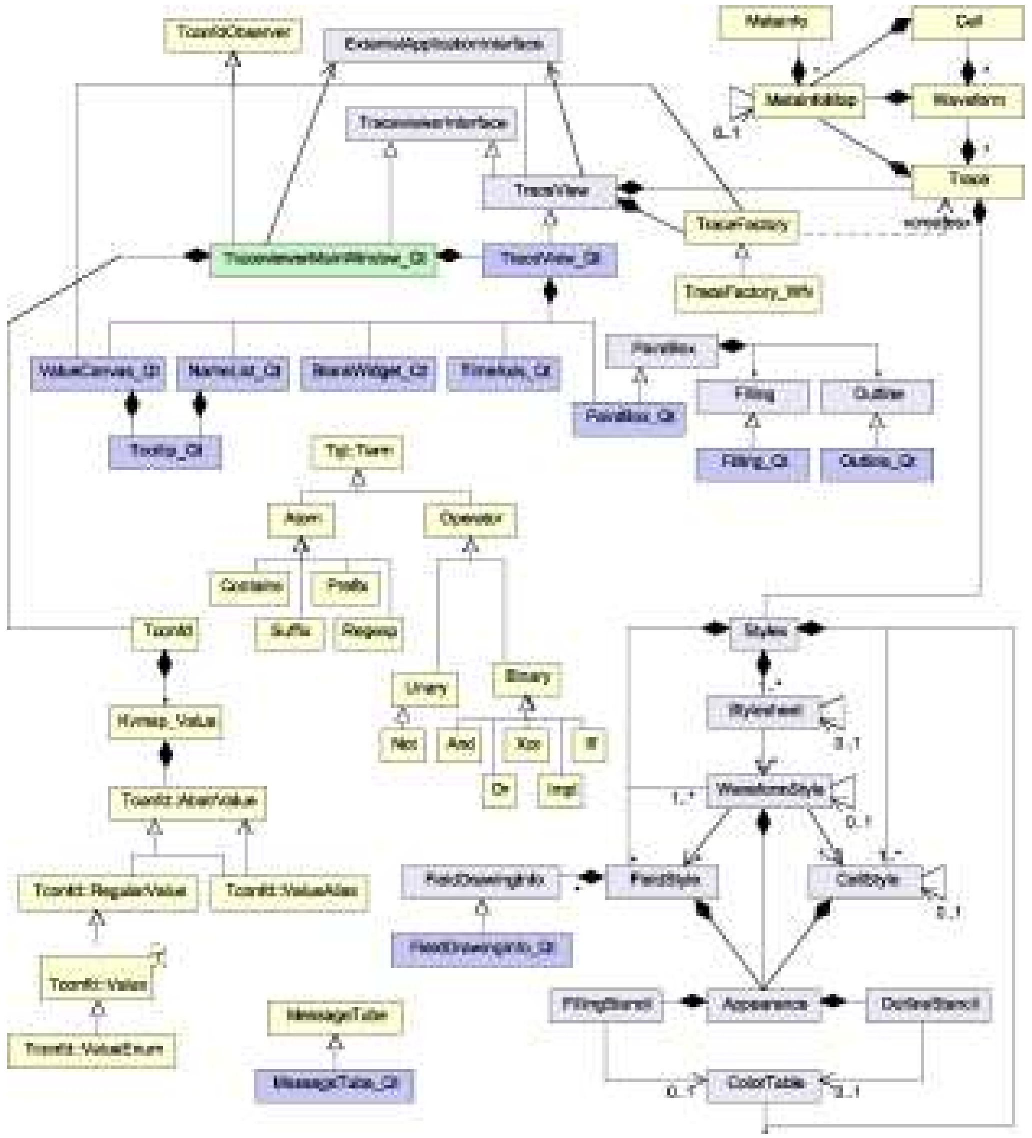




- **Example**: Timing Diagram Viewer
  Schumann et al. (2008)
- Architecture has four layers:
  - core, data layer
  - abstract view layer
  - toolkit-specific view layer/widget
  - application using widget

Example: Stereotypes for Documentation





- **Example**: Timing Diagram Viewer
  Schumann et al. (2008)
- Architecture has four layers:
  - core, data layer
  - abstract view layer
  - toolkit-specific view layer/widget
  - application using widget

Stereotype “=” layer “=” colour.

Other Examples

- Use stereotypes ‘Team$_1$’, ‘Team$_2$’, ‘Team$_3$’ and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.

- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).

- Use stereotypes ‘Server$_A$’, ‘Server$_B$’ to indicate where objects should be stored.

- Use stereotypes to label classes with states in the development process like “under development”, “submitted for testing”, “accepted”.

- etc. etc.

Other Examples

- Use stereotypes 'Team$_1$', 'Team$_2$', 'Team$_3$' and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.
- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).
- Use stereotypes 'Server$_A$', 'Server$_B$' to indicate where objects should be stored.
- Use stereotypes to label classes with states in the development process like "under development", "submitted for testing", "accepted".
- etc. etc.

Other Examples

- Use stereotypes ‘Team$_1$’, ‘Team$_2$’, ‘Team$_3$’ and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.
- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).
- Use stereotypes ‘Server$_A$’, ‘Server$_B$’ to indicate where objects should be stored.
- Use stereotypes to label classes with states in the development process like “under development”, “submitted for testing”, “accepted”.
- etc. etc.

Other Examples

- Use stereotypes ‘Team$_1$’, ‘Team$_2$’, ‘Team$_3$’ and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.

- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).

- Use stereotypes ‘Server$_A$’, ‘Server$_B$’ to indicate where objects should be stored.

- Use stereotypes to label classes with states in the development process like “under development”, “submitted for testing”, “accepted”.

- etc. etc.

Other Examples

- Use stereotypes 'Team$_1$', 'Team$_2$', 'Team$_3$' and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.

- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).

- Use stereotypes 'Server$_A$', 'Server$_B$' to indicate where objects should be stored.

- Use stereotypes to label classes with states in the development process like "under development", "submitted for testing", "accepted".

- etc. etc.

Other Examples

- Use stereotypes 'Team$_1$', 'Team$_2$', 'Team$_3$' and assign stereotype Team$_i$ to class C if Team$_i$ is responsible for class C.
- Use stereotypes to label classes with licensing information (e.g., LGPL vs. proprietary).
- Use stereotypes 'Server$_A$', 'Server$_B$' to indicate where objects should be stored.
- Use stereotypes to label classes with states in the development process like "under development", "submitted for testing", "accepted".
- etc. etc.

Necessary: a **common idea** of what each stereotype stands for.

(To be defined / agreed on by the team, not the job of the UML consortium.)

References

References

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

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.

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.