Contents & Goals

Last Lectures:
- Started to discuss “associations”, the general case.

This Lecture:
- **Educational Objectives**: Capabilities for following tasks/questions.
  - Cont’d: Please explain this class diagram with associations.
  - When is a class diagram a good class diagram?
  - What are purposes of modelling guidelines? (Example?)
  - Discuss the style of this class diagram.

- **Content**:
  - Treat “the rest”.
  - Where do we put OCL constraints?
  - Modelling guidelines, in particular for class diagrams (following [Ambler, 2005])
Recapitulation: Consider the following association:

\[ \langle r : \langle \text{role}_1 : C_1, \mu_1, P_1, \xi_1, \nu_1, o_1 \rangle, \ldots, \langle \text{role}_n : C_n, \mu_n, P_n, \xi_n, \nu_n, o_n \rangle \rangle \]

- **Association name** \( r \) and **role names/types** \( \text{role}_i \) induce extended system states \( \lambda \).
- **Multiplicity** \( \mu \) is considered in OCL syntax.
- **Visibility** \( \xi / \text{Navigability} \) \( \nu \): well-typedness.

Now the rest:
- **Multiplicity** \( \mu \): we propose to view them as constraints.
- **Properties** \( P_i \): even more typing.
- **Ownership** \( o \): getting closer to pointers/references.
- **Diamonds**: exercise.
Visibility

Not so surprising: Visibility of role-names is treated completely similar to visibility of attributes, namely by typing rules.

**Question:** given

```
1   1
C   D
\xi role
```

is the following OCL expression well-typed or not (wrt. visibility):

```
context C inv : self.role.x > 0
```

Basically same rule as before: (analogously for other multiplicities)

\[
(Assoc) \quad \frac{A, D \vdash expr_1 : \tau_C}{A, D \vdash \text{role}(expr_1) : \tau_D}, \quad \mu = 0..1 \text{ or } \mu = 1, \quad \xi = +, \text{ or } \xi = - \text{ and } C = D
\]

\[
(r : \ldots \langle \text{role} : D, \mu, \xi, \ldots \rangle, \ldots \langle \text{role} : C, \ldots \rangle, \ldots ) \in V
\]
Navigability

Navigability is similar to visibility: expressions over non-navigable association ends (ν = ×) are basically type-correct, but forbidden.

Question: given

is the following OCL expression well-typed or not (wrt. navigability):

\[ \text{context } D \text{ inv: } self.role.x > 0 \]

The standard says:
- '-' : navigation is possible
- '×' : navigation is not possible
- '>' : navigation is efficient

So: In general, UML associations are different from pointers/references!

But: Pointers/references can faithfully be modelled by UML associations.
Recapitulation: Consider the following association:

\[ \langle r : \langle \text{role}_1 : C_1, \mu_1, P_1, \xi_1, \nu_1, o_1 \rangle, \ldots, \langle \text{role}_n : C_n, \mu_n, P_n, \xi_n, \nu_n, o_n \rangle \rangle \]

- **Association name** \( r \) and **role names/types** \( \text{role}_i/C_i \) induce extended system states \( \lambda \).
- **Multiplicity** \( \mu \) is considered in OCL syntax.
- **Visibility** \( \xi \)/**Navigability** \( \nu \): well-typedness.

Now the rest:
- **Multiplicity** \( \mu \): we propose to view them as constraints.
- **Properties** \( P_i \): even more typing.
- **Ownership** \( o \): getting closer to pointers/references.
- **Diamonds**: exercise.
**Multiplicities as Constraints**

**Recall:** The multiplicity of an association end is a term of the form:
\[ \mu ::= * \mid N \mid N..M \mid N..* \mid \mu, \mu \quad (N, M \in \mathbb{N}) \]

**Proposal:** View multiplicities (except 0..1, 1) as additional invariants/constraints.

Recall: we can normalize each multiplicity \( \mu \) to the form
\[ \mu ::= N..2k \quad (N, k \in \mathbb{N}) \]

where \( N_i \leq N_{i+1} \) for \( 1 \leq i \leq 2k \), \( N_1, \ldots, N_{2k-1} \in \mathbb{N} \), \( N_{2k} \in \mathbb{N} \cup \{*, \} \).

**Example:**
\[ 31 \rightarrow 31..31 \]
\[ 0..* \rightarrow * \]
Multiplicities as Constraints

\[ \mu = N_1 \ldots N_2, \ldots, N_{2k-1} \ldots N_{2k} \]
where \( N_i \leq N_{i+1} \) for \( 1 \leq i \leq 2k \), \( N_1, \ldots, N_{2k-1} \in \mathbb{N} \), \( N_{2k} \in \mathbb{N} \cup \{\ast\} \).

Define \( \mu^{\text{COC}}(\text{role}) := \text{context } C \text{ inv : } \)
\[ (N_1 \leq \text{role} \rightarrow \text{size()} \leq N_2) \text{ or } \ldots \text{ or } (N_{2k-1} \leq \text{role} \rightarrow \text{size()} \leq N_{2k}) \]
for each \( \mu \neq 0.1, \mu \neq 1 \), \( \langle r : \ldots, \langle \text{role} : D, \mu, \ldots \rangle, \ldots, \langle \text{role}' : C, \ldots \rangle, \ldots \rangle \in V \text{ or } \langle r : \ldots, \langle \text{role}' : C, \ldots \rangle, \ldots, \langle \text{role} : D, \mu, \ldots \rangle, \ldots \rangle \in V, \text{role} \neq \text{role}' \).  

And define \( \mu^{\text{COC}}(\text{role}) := \text{context } C \text{ inv : } \not\text{oclIsUndefined}(\text{role}) \)
for each \( \mu = 1 \).

Note: in \( n \)-ary associations with \( n > 2 \), there is redundancy.

Multiplicities as Constraints of Class Diagram

Recall/Later:

\[ \mathcal{D} = \{CD_1, \ldots, CD_n\} \]
\[ \mathcal{P} = \text{signature } \mathcal{P}(\mathcal{D}) \]
\[ \text{invariants } \text{inv}(\mathcal{D}) \]
\[ \text{basic } \text{(classes and attributes)} \]
\[ \text{extended } \text{(visibility)} \]
\[ \text{distinguish} \]

From now on: \( \text{inv}(\mathcal{D}) = \{\text{constraints occurring in notes}\} \cup \{\mu^{\text{COC}}(\text{role}) | \langle r : \ldots, \langle \text{role} : D, \mu, \ldots \rangle, \ldots, \langle \text{role}' : C, \ldots \rangle, \ldots \rangle \in V \text{ or } \langle r : \ldots, \langle \text{role}' : C, \ldots \rangle, \ldots, \langle \text{role} : D, \mu, \ldots \rangle, \ldots \rangle \in V, \text{role} \neq \text{role}', \mu \notin \{0..1\} \} \).
### Multiplicities as Constraints Example

\[ \mu_{\text{OCL}} (\text{role}) = \text{context } C \text{ inv} : \]
\[ (N_1 \leq \text{role} \rightarrow \text{size}()) \leq N_2 \text{ or } \ldots \text{ or } (N_{2k-1} \leq \text{role} \rightarrow \text{size}()) \leq N_{2k} \]

**CD :**

```
role1 0..1 C
  v : Int
    1,17
role2 3..* C
```

**Inv(CD) =**

- \{ context \( C \text{ inv} : 4 \leq \text{role}_2 \rightarrow \text{size}()) \leq 4 \text{ or } 17 \leq \text{role}_2 \rightarrow \text{size}()) \leq 17 \}
- \{ context \( C \text{ inv} : \text{role}_2 \rightarrow \text{size}()) = 4 \text{ or } \text{role}_2 \rightarrow \text{size}()) = 17 \}
- \{ context \( C \text{ inv} : 3 \leq \text{role}_3 \rightarrow \text{size}()) \}

### Why Multiplicities as Constraints?

More precise, can’t we just use types? (cf. Slide 26)

- \( \mu = 0..1, \mu = 1 : \)
  - many programming languages have direct correspondences (the first corresponds to type pointer, the second to type reference) — therefore treated specially.
- \( \mu = * : \)
  - could be represented by a set data-structure type without fixed bounds — no problem with our approach, we have \( \mu_{\text{OCL}} = \text{true} \) anyway.
- \( \mu = 0..3 : \)
  - use array of size 4 — if model behaviour (or the implementation) adds 5th identity, we’ll get a runtime error, and thereby see that the constraint is violated. **Principally acceptable**, but: checks for array bounds everywhere...?
- \( \mu = 5..7 : \)
  - could be represented by an array of size 7 — but: few programming languages/data structure libraries allow lower bounds for arrays (other than 0). If we have 5 identities and the model behaviour removes one, this should be a violation of the constraints imposed by the **model**.
  - The implementation which does this removal is **wrong**. How do we see this...?
**Multiplicities Never as Types...?**

Well, if the **target platform** is known and fixed, **and** the target platform has, for instance,
- reference types,
- range-checked arrays with positions 0, \ldots, N,
- set types,
then we could simply **restrict** the syntax of multiplicities to

\[ \mu ::= 1 \mid 0..N \mid * \]

and don’t think about constraints
(but use the obvious 1-to-1 mapping to types)...

In general, **unfortunately**, we don’t know.

---

**Properties**

We don’t want to cover association **properties** in detail, only some observations (assume binary associations):

<table>
<thead>
<tr>
<th>Property</th>
<th>Intuition</th>
<th>Semantical Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique</td>
<td>one object has <strong>at most one</strong> $r$-link to a single other object</td>
<td><strong>current setting</strong></td>
</tr>
<tr>
<td>bag</td>
<td>one object may have <strong>multiple</strong> $r$-links to a single other object</td>
<td>have $\lambda(r)$ yield multi-sets</td>
</tr>
<tr>
<td>ordered, sequence</td>
<td>an $r$-link is a <strong>sequence</strong> of object identities (possibly including duplicates)</td>
<td>have $\lambda(r)$ yield sequences</td>
</tr>
</tbody>
</table>
Properties

We don’t want to cover association properties in detail, only some observations (assume binary associations):

<table>
<thead>
<tr>
<th>Property</th>
<th>Intuition</th>
<th>Semantical Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique</td>
<td>one object has at most one r-link to a single other object</td>
<td>current setting</td>
</tr>
<tr>
<td>bag</td>
<td>one object may have multiple r-links to a single other object</td>
<td>have $\lambda(r)$ yield multi-sets</td>
</tr>
<tr>
<td>ordered, sequence</td>
<td>an r-link is a sequence of object identities (possibly including duplicates)</td>
<td>have $\lambda(r)$ yield sequences</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>OCL Typing of expression $role(expr)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique</td>
<td>$\tau_D \rightarrow Set(\tau_C)$</td>
</tr>
<tr>
<td>bag</td>
<td>$\tau_D \rightarrow Bag(\tau_C)$</td>
</tr>
<tr>
<td>ordered, sequence</td>
<td>$\tau_D \rightarrow Seq(\tau_C)$</td>
</tr>
</tbody>
</table>

For subsets, redefines, union, etc. see [OMG, 2007a, 127].

Ownership

Intuitively it says:

Association $r$ is not a “thing on its own” (i.e. provided by $\lambda$), but association end ‘role’ is owned by $C$ (!).

(That is, it’s stored inside $C$ object and provided by $\sigma$).

So: if multiplicity of role is 0..1 or 1, then the picture above is very close to concepts of pointers/references.

Actually, ownership is seldom seen in UML diagrams. Again: if target platform is clear, one may well live without (cf. [OMG, 2007b, 42] for more details).

Not clear to me:
Recall: on some earlier slides we said, the extension of the signature is only to study associations in “full beauty”. For the remainder of the course, we should look for something simpler...

Proposal:

• from now on, we only use associations of the form

(i)

```
C ----> D
  ^       ^
  |       | 0..1
  |       | role

(ii)

C ----> D
  ^       ^
  |       | *
  |       | role
```

(And we may omit the non-navigability and ownership symbols.)

• Form (i) introduces role : C_{0,1}, and form (ii) introduces role : C*, in V.

• In both cases, role ∈ atr(C).

• We drop λ and go back to our nice σ with σ(u)(role) ⊆ \(\mathcal{P}(D)\).
Where Shall We Put OCL Constraints?

Numerous options:
(i) Additional documents.
(ii) Notes.
(iii) Particular dedicated places.

(i) Notes:
A UML note is a picture of the form

```
[ text ]
```

text can principally be everything, in particular comments and constraints.

Sometimes, content is explicitly classified for clarity:

```
OCL:
expr
```
OCL in Notes: Conventions

stands for

Where Shall We Put OCL Constraints?

(ii) **Particular dedicated places** in class diagrams: (behav. feature: later)

For simplicity, we view the above as an abbreviation for
**Invariants of a Class Diagram**

- Let $\mathcal{CD}$ be a class diagram.
- As we (now) are able to recognise OCL constraints when we see them, we can define

$$Inv(\mathcal{CD})$$

as the set $\{\varphi_1, \ldots, \varphi_n\}$ of OCL constraints occurring in notes in $\mathcal{CD}$ — after unfolding all abbreviations (cf. next slides).

- As usual: $Inv(\mathcal{D}) := \bigcup_{\mathcal{CD} \in \mathcal{D}} Inv(\mathcal{CD})$.
- **Principally clear:** $Inv(\cdot)$ for any kind of diagram.

---

**Invariant in Class Diagram Example**

If $\mathcal{D}$ consists of only $\mathcal{CD}$ with the single class $C$, then
- $Inv(\mathcal{D}) = Inv(\mathcal{CD}) = \{\varphi\}$
**Definition.** Let $\mathcal{D}$ be a set of class diagrams. We say, the **semantics** of $\mathcal{D}$ is the signature it induces and the set of OCL constraints occurring in $\mathcal{D}$, denoted

$$[\mathcal{D}] := (\mathcal{S}(\mathcal{D}), \text{Inv}(\mathcal{D})).$$

Given a structure $\mathcal{S}$ of $\mathcal{D}$ (and thus of $\mathcal{C}$), the class diagrams describe the system states $\Sigma_{\mathcal{C}}$. Of those, some satisfy $\text{Inv}(\mathcal{C})$ and some don’t. We call a system state $\sigma \in \Sigma_{\mathcal{C}}$ **consistent** if and only if $\sigma \models \text{Inv}(\mathcal{C})$.

**In pictures:**

A set of class diagrams $\mathcal{D}$ describes the **structure** of system states.

Together with the invariants it can be used to state:

- **Pre-image**: Dear programmer, please provide an implementation which uses only system states that satisfy $\text{Inv}(\mathcal{C})$.

- **Post-image**: Dear user/maintainer, in the existing system, only system states which satisfy $\text{Inv}(\mathcal{C})$ are used.

(The exact meaning of “use” will become clear when we study behaviour — intuitively: the system states that are reachable from the initial system state(s) by calling methods or firing transitions in state-machines.)

**Example**: highly abstract model of traffic lights controller.
Constraints vs. Types

Find the 10 differences:

<table>
<thead>
<tr>
<th>C</th>
<th>(x : \text{Int} { x = 3 \lor x &gt; 17 } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x : T )</td>
<td>(\mathcal{D}(T) = {3} \cup {n \in \mathbb{N}</td>
</tr>
</tbody>
</table>

- \(x = 4\) is well-typed in the left context, a system state satisfying \(x = 4\) violates the constraints of the diagram.
- \(x = 4\) is not even well-typed in the right context, there cannot be a system state with \(\sigma(u)(x) = 4\) because \(\sigma(u)(x)\) is supposed to be in \(\mathcal{D}(T)\) (by definition of system state).

Rule-of-thumb:

- If something “feels like” a type (one criterion: has a natural correspondence in the application domain), then make it a type.
- If something is a requirement or restriction of an otherwise useful type, then make it a constraint.

References
References

