Software Design, Modelling and Analysis in UML

Lecture 20: Inheritance

2016-02-04

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany


Last Lecture:
- Firedset, Cut
- Automaton construction
- Transition annotations

This Lecture:

- Educational Objectives: Capabilities for following tasks/questions.
  - What’s the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset / uplink semantics of inheritance?
  - What’s the effect of inheritance on LSCs, State Machines, System States?

- Content:
  - Inheritance in UML: concrete syntax
  - Liskov Substitution Principle — desired semantics
  - Two approaches to obtain desired semantics
Inheritance: Syntax
A **signature with inheritance** is a tuple

\[ \mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}, F, mth, \triangleleft) \]

where

- \((\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E})\) is a signature with signals and behavioural features \((F/mth\) are methods, analogous to \(V/atr\) attributes), and
- \(\triangleleft \subseteq (\mathcal{C} \times \mathcal{C}) \cup (\mathcal{E} \times \mathcal{E})\)
  is an **acyclic generalisation** relation, i.e. \(C \triangleleft^+ C\) for no \(C \in \mathcal{C}\).

In the following (for simplicity), we assume that all attribute (method) names are of the form \(C::v\) and \(C::f\) for some \(C \in \mathcal{C} \cup \mathcal{E}\) ("fully qualified names").

Read \(C \triangleleft D\) as...

- \(D\) **inherits** from \(C\),
- \(C\) is a **generalisation** of \(D\),
- \(D\) is a **specialisation** of \(C\),
- \(C\) is a **super-class** of \(D\),
- \(D\) is a **sub-class** of \(C\),
- ...
Definition.

(i) For classes $C_0, C_1, D \in \mathcal{C}$, we say $D$ inherits from $C_0$ via $C_1$ if and only if there are $C_0^1, \ldots, C_0^n, C_1^1, \ldots, C_1^m \in \mathcal{C}$, $n, m \geq 0$, s.t.

\[
C_0 \triangleleft C_0^1 \triangleleft \ldots \triangleleft C_0^n \triangleleft C_1 \triangleleft C_1^1 \triangleleft \ldots \triangleleft C_1^m \triangleleft D.
\]

(ii) We use $\triangleleft^*$ to denote the reflexive, transitive closure of $\triangleleft$. 

Inheritance: Concrete Syntax

**Common graphical representations** (of $\sqsubseteq \{(C, D_1), (C, D_2)\}$):

![Graphical representations of inheritance](image)

**Mapping** Concrete to Abstract Syntax by Example:

$\varphi = \{ (C_0, C_1), (C_1, C_2), (D, C_2) \}$

**Note:** we can have *multiple inheritance.*
Inheritance: Desired Semantics
There is a classical description of what one expects from sub-types, which is closely related to inheritance in object-oriented approaches:

The principle of type substitutability Liskov (1988); Liskov and Wing (1994) (Liskov Substitution Principle (LSP)).

“If for each object \( o_1 \) of type \( S \)
there is an object \( o_2 \) of type \( T \)
such that for all programs \( P \) defined in terms of \( T \)
the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \)
then \( S \) is a subtype of \( T \).”

In other words: Fischer and Wehrheim (2000)

“An instance of the sub-type shall be usable
whenever an instance of the supertype was expected,
without a client being able to tell the difference.”
Subtyping: Example

**Teacher**: 0, 1

**Student**: 0, 1

- **Genius**
- **Polite**
- **Clown**

**GenStWorker**: workload : Int

**SM**

- **SM_{Teacher}**
  - s1
  - s2

- **SM_{Student}**
  - s1
  - s2

- **SM_{Genius}**
  - s1
  - s2

- **SM_{Polite}**
  - s1
  - s2

- **SM_{Clown}**
  - s1
  - s2

- **SM_{GenStWorker}**
  - s1
  - s2
  - s3

**Tasks**

- **GoodAns**
- **WrongAns**
- **Silence**
- **StupidJoke**

**Att**:

- **Int**

**GenStWorker**

- **task : Int**
Domain Inclusion Semantics
A **domain inclusion structure** $\mathcal{D}$ for signature $\mathcal{S} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}, F, mth, \triangleleft)$

- [as before] maps types, classes, associations to domains,
- [for completeness] maps methods to transformers,
- [as before] has infinitely many object identities per class in $\mathcal{D}(D), \mathcal{D} \in \mathcal{C}$,
- [changed] the identities of a super-class comprise all identities of sub-classes, i.e.

$$\forall C \triangleleft D \in \mathcal{C} : \mathcal{D}(D) \subsetneq \mathcal{D}(C)$$

and identities of instances of classes not (transitively) related by generalisation are disjoint, i.e. $C \not\triangleleft^+ D$ and $D \not\triangleleft^+ C$ implies $\mathcal{D}(C) \cap \mathcal{D}(D) = \emptyset$.

**Note:** the old setting coincides with the special case $\triangleleft = \emptyset$. 

![Diagram](image-url)
A **system state** of $\mathcal{I}$ wrt. (domain inclusion structure) $\mathcal{D}$ is a **type-consistent** mapping

$$\sigma : \mathcal{D}(\mathcal{C}) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{T}) \cup \mathcal{D}(\mathcal{C}_{0,1}) \cup \mathcal{D}(\mathcal{C}^*))$$

that is, for all $u \in \text{dom}(\sigma) \cap \mathcal{D}(\mathcal{C})$,

- **[as before]** $\sigma(u)(v) \in \mathcal{D}(T)$ if $v : T$,
- **[changed]** $\sigma(u), u \in \mathcal{D}(\mathcal{C})$, has values for all attributes of $\mathcal{C}$ and all of its superclasses, i.e.

$$\text{dom}(\sigma(u)) = \bigcup_{C_0 \prec^* C} \text{atr}(C_0).$$

**Example:**

<table>
<thead>
<tr>
<th>$u_1 : \mathcal{C}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{C} : x = 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$u_2 : \mathcal{D}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{D} : x = 1$</td>
</tr>
<tr>
<td>$\mathcal{D} : y = 2$</td>
</tr>
</tbody>
</table>

**Note:** the old setting still coincides with the special case $\preceq = \emptyset$. 
OCL Syntax and Typing

- Recall (part of the) OCL syntax and typing ($C, D \in \mathcal{C}, v, r \in V$)

\[
expr ::= \begin{array}{ll}
v(expr_1) & : \tau_C \rightarrow T(v), \quad \text{if } v : T \in \text{atr}(C), \quad T \in \mathcal{I} \\
r(expr_1) & : \tau_C \rightarrow \tau_D, \quad \text{if } r : D_{0,1} \in \text{atr}(C) \\
r(expr_1) & : \tau_C \rightarrow \text{Set}(\tau_D), \quad \text{if } r : D_\star \in \text{atr}(C)
\end{array}
\]

The syntax **basically** stays the same:

\[
expr ::= \begin{array}{ll}
C::v(expr_1) & : \tau_C \rightarrow T(v), \quad \text{if } C::v : T \in \text{atr}(C), \quad T \in \mathcal{I} \\
\ldots
\end{array}
\]

\[
\begin{array}{ll}
v(expr_1) & : \tau_C \rightarrow T(v), \\
r(expr_1) & : \tau_C \rightarrow \tau_D, \\
r(expr_1) & : \tau_C \rightarrow \text{Set}(\tau_D),
\end{array}
\]

but **typing rules change**: we require a unique biggest superclass $C_0 \downarrow^* C \in \mathcal{C}$ with, e.g., $v \in \text{atr}(C_0)$ and for this $v$ we have $v : T$.

**Example:**

\begin{align*}
\text{context } C & \text{' inv } C::x > 0 \\
\text{context } D & \text{' inv } C::x > 0 \\
\text{context } D & \text{' inv } x > 0
\end{align*}

**Note:** the old setting still coincides with the special case $\triangleleft = \emptyset$. 
Example:

<table>
<thead>
<tr>
<th></th>
<th>$v_1 &lt; 0$</th>
<th>$v_2 &lt; 0$</th>
<th>$v_3 &lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>context $C$ inv:</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>context $D$ inv:</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$n.v_1 &lt; 0$</td>
<td>$n.v_2 &lt; 0$</td>
<td>$n.v_3 &lt; 0$</td>
</tr>
<tr>
<td>context $B$ inv:</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

E.g. $v(\ldots (self) \ldots)$ is well-typed

- if $v$ is public, or
- if $v$ is private, and $self : \tau_C$ and $v \in atr(C)$, or
- if $v$ is protected, and $self : \tau_C$ and $D \triangleleft^* C$ (unique, biggest) and $v \in atr(D)$. 
\[ I_{DI}[expr](\sigma) := I[\text{Normalise}(expr)](\sigma) \]

using the same \textbf{textual} definition of \( I \) that we have.
Expression Normalisation

Normalise:

- Given expression \( v(\ldots(w)\ldots) \) with \( w : \tau_D \),
- normalise \( v \) to (= replace by) \( C::v \),
- where \( C \) is the unique most special more general class with \( C::v \in \text{attr}(C) \), i.e.

\[
\forall C \triangleleft^* C_0 \triangleleft^* D \bullet C_0 = C.
\]

Note: existence of such an \( C \) is guaranteed by (the new) OCL well-typedness.

Example:

- context \( D \ inv : x < 0 \) \( \triangleright \) context \( D \ inv : \mathcal{C}::x > 0 \)
- context \( C \ inv : x < 0 \) \( \triangleright \) \( \triangleright \) \( \triangleright \)
- context \( A \ inv : x < 0 \) \( \triangleright \) \( \triangleright \) \( \triangleright \)
- context \( D \ inv : n < 0 \) \( \triangleright \) \( \triangleright \) \( \triangleright \)
- context \( C \ inv : n < 0 \) \( \triangleright \)
- context \( D \ inv : A::x < 0 \) \( \triangleright \) \( \triangleright \)
OCL Example

\(\sigma:\)

\[
\begin{align*}
    u_1 : A & \quad A::x = 0 \\
    u_2 : C & \quad A::x = 1 \quad C::x = 27 \\
    u_3 : D & \quad A::x = 2 \quad C::x = 13
\end{align*}
\]

- \(I[context \ D \ inv : A::x < 0](\sigma, \{self \mapsto u_3\})\)

\[
\llbracket \sigma(u_3)(A::x), 0 \rrbracket = \llbracket 2, 0 \rrbracket = false
\]

- \(I[context \ D \ inv : x < 0](\sigma, \{self \mapsto u_3\})\)

\[
\llbracket \sigma(u_3)(C::x), 0 \rrbracket = \llbracket 13, 0 \rrbracket = false
\]

\(I[v(expr_1)](\sigma, \beta) := \begin{cases} 
\sigma(u_1)(v), & \text{if } u_1 \in \text{dom}(\sigma) \\
\bot, & \text{otherwise}
\end{cases}\)
Excursus: Late Binding of Behavioural Features
What transformer applies in what situation? (Early (compile time) binding.)

<table>
<thead>
<tr>
<th>f not overridden in D</th>
<th>f overridden in D</th>
<th>value of someC/someD</th>
</tr>
</thead>
<tbody>
<tr>
<td>C::f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>C::f()</td>
<td>D::f()</td>
<td></td>
</tr>
<tr>
<td>C::f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>D::f()</td>
<td></td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
</tbody>
</table>

What one could want is something different: (Late binding.)

<table>
<thead>
<tr>
<th>someC -&gt; f()</th>
<th>C::f()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>someD -&gt; f()</td>
<td>D::f()</td>
<td></td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>C::f()</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
  C
  |    
  v    
 someC -> f() C::f()  
  |    
  v    
 someD -> f() D::f()  
```

```
  C
  |    
  v    
 someC -> f() C::f()  
  |    
  v    
 someD -> f() D::f()  
```
In the standard, Section 11.3.10, “CallOperationAction”:

“Semantic Variation Points
The mechanism for determining the method to be invoked as a result of a call operation is unspecified.” (OMG, 2007, 247)

In C++,
- methods are by default “(early) compile time binding”,
- can be declared to be “late binding” by keyword “virtual”,
- the declaration applies to all inheriting classes.

In Java,
- methods are “late binding”;
- there are patterns to imitate the effect of “early binding”

Note: late binding typically applies only to methods, not to attributes.
(But: getter/setter methods have been invented recently.)
Behaviour (Inclusion Semantics)
Semantics of Method Calls

- **Non late-binding**: by normalisation.

- **Late-binding**:
  
  Construct a **method call** transformer, which looks up the method transformer corresponding to the class we are an instance of.
Transformers also basically remain the same, e.g. [VL 12, p. 18]

\[
update(expr_1, v, expr_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon)
\]

with

\[
\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I_{DI}[expr_2](\sigma)]]
\]

where \( u = I_{DI}[expr_1](\sigma) \) — after normalisation, e.g. assume \( v \) qualified.
Inheritance and State-Machines: Example

\[ \langle \text{signal, env} \rangle \]

\[ E \]

\[ \langle \text{signal, env} \rangle \]

\[ F \]

\[ \text{SM}_A: \]

\[ /n!F \]

\[ s_1 \rightarrow s_2 \]

\[ \text{SM}_D: \]

\[ E/ \]

\[ s_1 \rightarrow s_2 \]

\[ \text{C} \]

\[ \text{A} \]

\[ n \]

\[ 0, 1 \]

\[ \text{D} \]

\[ u_1 : A \]

\[ st = s_1 \]

\[ stable = 0 \]

\[ n \]

\[ u_2 : D \]

\[ st = s_1 \]

\[ stable = 1 \]

\[ (\emptyset, (u_2, f_{1+})) \]

\[ \text{v}_1 \]

\[ \text{v}_1 : A \]

\[ \frac{st = s_2}{stable = 1} \]

\[ (f, \emptyset) \]

\[ \text{v}_2 \]

\[ \text{v}_2 : D \]

\[ \frac{st = s_2}{stable = 1} \]

\[ \epsilon : (f_1 u_2) \]

\[ \epsilon : \epsilon \]
(ii) Dispatch

\[(\sigma, \varepsilon) \xrightarrow{\text{cons}, \text{Snd}}_{u} (\sigma', \varepsilon')\]

if

- \(u \in \text{dom}(\sigma) \cap \mathcal{D}(C) \land \exists u_E \in \mathcal{D}(E) : u_E \in \text{ready}(\varepsilon, u)\)
- \(u\) is stable and in state machine state \(s\), i.e. \(\sigma(u)(\text{stable}) = 1\) and \(\sigma(u)(\text{st}) = s\),
- a transition is enabled, i.e.

\[
\exists (s, F, \text{expr, act, } s') \in \rightarrow (SM_C) : F = E \land I[\text{expr}] (\tilde{\sigma}, u) = 1
\]

where \(\tilde{\sigma} = \sigma[u.\text{params}_E \mapsto u_E]\).

and

- \((\sigma', \varepsilon')\) results from applying \(t_{\text{act}}\) to \((\sigma, \varepsilon)\) and removing \(u_E\) from the ether, i.e.

\[
(\sigma'', \varepsilon') \in t_{\text{act}}[u](\tilde{\sigma}, \varepsilon \oplus u_E),
\]

\[
\sigma' = (\sigma''[u.\text{st} \mapsto s', u.\text{stable} \mapsto b, u.\text{params}_E \mapsto \emptyset])|\mathcal{D}(\varepsilon)\backslash\{u_E\}
\]

where \(b\) depends (see (i))

- Consumption of \(u_E\) and the side effects of the action are observed, i.e.

\[
\text{cons} = \{u_E\}, \quad \text{Snd} = \text{Obst}_{\text{act}}[u](\tilde{\sigma}, \varepsilon \oplus u_E).
\]
Inheritance and Interactions

\[ \exists \beta, \beta(a) \in \mathcal{D}(A), \beta(c) \in C' \cdot \ldots \]

\( E^i \)

\( \langle \text{signal, env} \rangle \)

\( E \)

\( F \)

\( a : A \)

\( c : C \)

\( E \)

\( C \)

\( n \)

\( 0, 1 \)

\( A \)

\( D \)

\( \langle \text{signal, env} \rangle \)

\( E \)

\( F \)

\( a : A \)

\( u_1 \)

\( u_2 \)

\( v_1 ; D \)

\( (\sigma, u, \text{on}, \text{sw}) \)

\( E^i_{a,c} \)

\( \checkmark \)
Domain Inclusion vs. Uplink Semantics
**Wanted**: a formal representation of “if \( C \triangleleft^* D \) then \( D \) ‘is a’ \( C \)’, that is,

(i) \( D \) has the same attributes and behavioural features as \( C \), and (ii) \( D \) objects (identities) can replace \( C \) objects.

**Two approaches** to semantics:

- **Domain-inclusion** Semantics

  (more theoretical)

- **Uplink** Semantics

  (more technical)
References
References


