Software Design, Modelling and Analysis in UML

Lecture 21: Inheritance

2015-02-05

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:
- Live Sequence Charts Semantics

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What’s the Liskov Substitution Principle?
  - What is late/early binding?
  - What is the subset, what the 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
Motivations for Generalisation

- Re-use,
- Sharing,
- Avoiding Redundancy,
- Modularisation,
- Separation of Concerns,
- Abstraction,
- Extensibility,
- ...

→ See textbooks on object-oriented analysis, development, programming.
Abstract Syntax

Recall: a signature (with signals) is a tuple $\mathcal{S} = (T, C, V, atr, E)$.

Now (finally): extend to $\mathcal{S} = (T, C, V, atr, E, mth, \triangleright)$ where $F/mth$ are methods, analogously to attributes and

$\triangleright \subseteq ((C \setminus E) \times (C \setminus E)) \cup (E \times E)$

is a generalisation relation such that $C \triangleright C$ for no $C \in \mathcal{C}$ (“acyclic”).

$C \triangleright D$ reads as

- $C$ is a generalisation of $D$,
- $D$ is a specialisation of $C$,
- $D$ inherits from $C$,
- $D$ is a sub-class of $C$,
- $C$ is a super-class of $D$,
- ...
Definition. Given 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 $C_0^1, \ldots, C_0^n, C_1^1, \ldots, C_1^m \in \mathcal{C}$ such that

$$C_0 \lessdot C_0^1 \lessdot \ldots \lessdot C_0^n \lessdot C_1 \lessdot C_1^1 \lessdot \ldots \lessdot C_1^m \lessdot D.$$ 

We use ‘$\lessdot$’ to denote the reflexive, transitive closure of ‘$\lessdot$’.

In the following, we assume

- that all attribute (method) names are of the form $$C::v, \quad C \in \mathcal{C} \cup \mathcal{E} \\ C::f, \quad C \in \mathcal{C},$$
- that we have $C::v \in \text{attr}(C)$ resp. $C::f \in \text{mth}(C)$ if and only if $v(f)$ appears in an attribute (method) compartment of $C$ in a class diagram.

Extend Typing Rules
Recall: With extension for visibility we obtained

\[ v(w) : \tau_C \rightarrow \tau(v) \quad \langle v : \tau, \xi, expr_0, P_C \rangle \in atr(C), \ w : \tau_C \]

\[ v(expr_1(w)) : \tau_{C_2} \rightarrow \tau(v) \quad \langle v : \tau, \xi, expr_0, P_C \rangle \in atr(C_2), \]

\[ expr_1(w) : \tau_{C_2}, \ w : \tau_{C_1}, \text{ and } C_1 = C_2 \text{ or } \xi = + \]

Now:

\[ v(w) : \tau_C \rightarrow \tau(v) \quad \langle v : \tau, \xi, expr_0, P_C \rangle \in atr(C), \ w : \tau_{C_1}, \tau_C \preceq \tau_{C_1} \]

\[ v(expr_1(w)) : \tau_{C_2} \rightarrow \tau(v) \quad \langle v : \tau, \xi, expr_0, P_C \rangle \in atr(C_2), \]

\[ expr_1(w) : \tau_{C_2}, \ w : \tau_{C_1}, \text{ and } (C_1 = C_2 \text{ or } \xi = + \text{ or } (C_2 \preceq C_1 \text{ and } \xi = \#)) \]

Inheritance: System States
System States

Wanted: a formal representation of “if $C \leq D$ then $D$ is a $C_i$”, i.e.,
(i) $D$ has the same attributes as $C$, and
(ii) $D$ objects (identities)
can be used in any context where $C$ objects can be used.

We’ll discuss two approaches to semantics:

• **Domain-inclusion** Semantics (more theoretical)

• **Uplink** Semantics (more technical)

Domain Inclusion Semantics
Domain Inclusion Semantics: Idea

Let $\mathcal{S} = (T, C, V, atr, E, F, mth, \triangleleft)$ be a signature.

Now a structure $\mathcal{D}$

- [as before] maps types, classes, associations to domains,
- [for completeness] methods to transformers,
- [as before] identities of instances of classes not (transitively) related by generalisation are disjoint,
- [changed] the identities of a super-class comprise all identities of sub-classes, i.e.
  $$\forall C \in C : \mathcal{D}(C) \supseteq \bigcup_{C\subset D} \mathcal{D}(D).$$

Note: the old setting coincides with the special case $\triangleleft = \emptyset$. 
Domain Inclusion System States

Now: a system state of $\mathcal{I}$ wrt. $\mathcal{D}$ is a type-consistent mapping

$$\sigma : \mathcal{D}(C) \rightarrow (V \rightarrow (\mathcal{D}(\mathcal{I}) \cup \mathcal{D}(C_{0,1}) \cup \mathcal{D}(C_s)))$$

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

- [as before] $\sigma(u)(v) \in \mathcal{D}(\tau)$ if $v : \tau$, $\tau \in \mathcal{I}$ or $\tau \in \{C_s, C_{0,1}\}$.
- [changed] $\text{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} \text{atr}(C_0)$.

Example:

Note: the old setting still coincides with the special case $\varnothing = \emptyset$.

Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{M}, \mathcal{I})$ be a UML model, and $\mathcal{D}$ a structure.
- We (continue to) say $\mathcal{M} \models \text{expr}$ for context $C$ inv : $\text{expr}_0 \in \text{Inv}(\mathcal{M})$ iff

$$\forall \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in [\mathcal{M}] \forall i \in \mathbb{N} \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) : I[\text{expr}_0](\sigma_i, \{\text{self} \mapsto u\}) = 1.$$

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\text{Inv}(\mathcal{M})$.

Example: e:

$$\text{dom}(\sigma) \cap \mathcal{D}(\text{swap}) = \{v_1, v_2\}$$
Transformers (Domain Inclusion)

- Transformers also remain the same, e.g. [VL 12, p. 18]

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

with

\[
\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[expr_2]\langle \sigma \rangle]]
\]

where \( u = I[expr_1]\langle \sigma \rangle \).

Inheritance and State Machines: Triggers

- Wanted: triggers shall also be sensitive for inherited events, sub-class shall execute super-class’ state-machine (unless overridden).

\[
\text{(sigma, epsilon)} \xrightarrow{\text{cons, Snd}} (sigma', epsilon') \text{ if}
\]

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

\[
\exists (s, F, expr, act, s') \in (SM_C) : F = E \land I[expr]\langle \hat{\sigma} \rangle = 1
\]

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

and

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

\[
(\sigma'', \varepsilon') = t_{act}(\hat{\sigma}, \varepsilon \ominus u_E),
\]

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

where \( b \) depends:
- If \( u \) becomes stable in \( s' \), then \( b = 1 \). It does become stable if and only if there is no transition without trigger enabled for \( u \) in \((\sigma', \varepsilon')\).
- Otherwise \( b = 0 \).
Similar to satisfaction of OCL expressions above:

- An instance line stands for all instances of \( C \) (exact or inheriting).
- Satisfaction of event observation has to take inheritance into account, too, so we have to fix, e.g.

\[
\sigma, \text{cons, } \text{Snd} \models \beta_{x,y} \quad E_x^! \text{ if and only if } \beta(x) \text{ sends an } F\text{-event to } \beta(y) \text{ where } E \preceq F.
\]

- \( C \)-instance line also binds to \( C' \)-objects.

**Uplink Semantics**
Uplink Semantics: Idea

context $s_1, s_2 : \text{Sensor}$ $\text{inv} : v < 0$

\[
\begin{align*}
&u_1 : \text{SmokeSensor} \\
&\quad \text{Label} = "51.3.1" \quad s = 3
\end{align*}
\]

\[
\begin{align*}
&u_2 : \text{TempSensor} \\
&\quad \text{Label} = "51.3.17" \quad t = 19.7
\end{align*}
\]

* Idea:

- Continue with the existing definition of *structure*, i.e. disjoint domains for identities.
- Have an *implicit association* from the child to each parent part (similar to the implicit attribute for stability).

\[
\begin{align*}
C &\quad x : \text{Int} \\
D
\end{align*}
\]

- Apply (a different) pre-processing to make appropriate use of that association, e.g. rewrite (C++)

\[
x = 0;
\]

in $D$ to

\[
\text{uplink}_C \rightarrow x = 0;
\]
Pre-Processing for the Uplink Semantics

- For each pair $C \prec D$, extend $D$ by a (fresh) association
  \[ \text{uplink}_C : C \text{ with } \mu = [1, 1], \xi = + \]
  
  (Exercise: public necessary?)

- Given expression $v$ (or $f$) in the context of class $D$,
  
  - let $C$ be the smallest class wrt. “$\preceq$” such that
    - $C \preceq D$, and
    - $C::v \in \text{atr}(D)$
  
  - then there exists (by definition) $C \prec C_1 \prec \ldots \prec C_n \prec D$,
  
  - normalise $v$ to (= replace by)
    \[ \text{uplink}_{C_n} \rightarrow \cdots \rightarrow \text{uplink}_{C_1}.C::v \]

- If no (unique) smallest class exists,
  the model is considered not well-formed; the expression is ambiguous.

Uplink Structure, System State, Typing

- Definition of structure remains unchanged.

- Definition of system state remains unchanged.

- Typing and transformers remain unchanged — the preprocessing has put everything in shape.
Satisfying OCL Constraints (Uplink)

- Let $\mathcal{M} = (\mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{P})$ be a UML model, and $\mathcal{D}$ a structure.
- We (continue to) say
  \[ \mathcal{M} \models expr \]
  for
  \[ \text{context } C \quad \text{inv} : \text{expr}_0 \in \text{Inv}(\mathcal{M}) \]
  \[ = \text{expr} \]
  if and only if
  \[ \forall \pi = (\sigma_i)_{i \in \mathbb{N}} \in [\mathcal{M}] \]
  \[ \forall i \in \mathbb{N} \]
  \[ \forall u \in \text{dom}(\sigma_i) \cap \mathcal{D}(C) : \]
  \[ I[\text{expr}_0](\sigma_i, \{ \text{self} \mapsto u \}) = 1. \]
- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\text{Inv}(\mathcal{M})$.

Transformers (Uplink)

- What has to change is the create transformer:
  \[ \text{create}(C, expr, v) \]
- Assume, $C$'s inheritance relations are as follows.
  \[ C_{1,1} \subset \ldots \subset C_{1,n_1} \subset C, \]
  \[ \ldots \]
  \[ C_{m,1} \subset \ldots \subset C_{m,n_m} \subset C. \]
- Then, we have to
  - create one fresh object for each part, e.g.
    \[ u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m}, \]
  - set up the uplinks recursively, e.g.
    \[ \sigma(u_{1,2})(\text{uplink}_{C_{1,1}}) = u_{1,1}. \]
  - And, if we had constructors, be careful with their order.
Cast-Transformers

- C c;
- D d;

- **Identity upcast** (C++):
  - C* cp = &d;  // assign address of ‘d’ to pointer ‘cp’

- **Identity downcast** (C++):
  - D* dp = (D*)cp;  // assign address of ‘d’ to pointer ‘dp’

- **Value upcast** (C++):
  - *c = *d;  // copy attribute values of ‘d’ into ‘c’, or,
    // more precise, the values of the C'-part of ‘d’
Casts in Domain Inclusion and Uplink Semantics

### Domain Inclusion

<table>
<thead>
<tr>
<th>C* cp = &amp;d;</th>
<th>( \text{easy: immediately compatible (in underlying system state) because } &amp;d \text{ yields an identity from } \mathcal{D}(D) \subset \mathcal{D}(C). )</th>
</tr>
</thead>
</table>

### Uplink

<table>
<thead>
<tr>
<th>D* dp = (D*)cp;</th>
<th>( \text{easy: the value of cp is in } \mathcal{D}(D) \cap \mathcal{D}(C) \text{ because the pointed-to object is a } D. ) (Otherwise, error condition.)</th>
</tr>
</thead>
</table>

| c = d; | \( \text{bit difficult: set (for all } C \preceq D \text{)} \) |

### Identity Downcast with Uplink Semantics

- **Recall** (C++): \( D \text{ d; } C* \text{ cp = &d; } D* \text{ dp = (D*)cp;} \)
- **Problem**: we need the identity of the \( D \) whose \( C \)-slice is denoted by \( cp \).
- **One technical solution**:
  - Give up disjointness of domains for **one additional type** comprising all identities, i.e. have
  \[
  \text{all } \in \mathcal{I}, \quad \mathcal{D} \text{(all)} = \bigcup_{C \in \mathcal{I}} \mathcal{D}(C)
  \]
  - In each \( \preceq \)-minimal class have associations “mostspec” pointing to **most specialised** slices, plus information of which type that slice is.
  - Then **downcast** means, depending on the mostspec type (only finitely many possibilities), **going down and then up** as necessary, e.g.

  ```
  switch(mostspec_type){
  case C:
    dp = cp -> mostspec -> uplink_{D_n} -> ... -> uplink_{D_1} -> uplink_{D};
  ...
  ```
Domain Inclusion vs. Uplink: Differences

• **Note:** The uplink semantics views inheritance as an abbreviation:
  
  • We only need to touch transformers (create) — and if we had constructors, we didn’t even needed that (we could encode the recursive construction of the upper slices by a transformation of the existing constructors.)
  
  • **So:**
    
    • Inheritance doesn’t add expressive power.
    • And it also doesn’t improve conciseness soo dramatically.

As long as we’re “early binding”, that is...

Domain Inclusion vs. Uplink: Motivations

• **Exercise:**

  What’s the point of

  • having the tedious adjustments of the theory if it can be approached technically?
  
  • having the tedious technical pre-processing if it can be approached cleanly in the theory?
More Interesting: Behaviour

Example: Behaviour of Kinds of Students
There is a classical description of what one expects from sub-types, which in the OO domain is closely related to inheritance:

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."

So, what’s “usable”? Who’s a “client”? And what’s a “difference”?

“...shall be usable...” for UML
Given:

\[
\begin{align*}
C & \colon \text{itsC1} \colon \text{Int}, f(\text{Int}) : \text{Int} \\
D & \colon \text{itsD1} \colon \text{Int}, f(\text{Int}) : \text{Int} \\
E & \colon \text{itsC1} \colon \text{Bool}, f(\text{Float}) : \text{Int} \\
F & \colon \text{itsD2} \colon \text{Float}, f(\text{Float}) : \text{Int}
\end{align*}
\]

Wanted:

- \(x > 0\) also **well-typed** for \(D_1\)
- assignment \(\text{itsC1} := \text{itsD1}\) being **well-typed**
- \(\text{itsC1} . x = 0, \text{itsC1} . f(0), \text{itsC1} ! F\)  
being well-typed (and doing the right thing).

Approach:

- Simply define it as being well-typed,  
  adjust system state definition to do the right thing.

---

**Static Typing for Methods**

\[
\begin{align*}
C' & \colon \text{itsC1} \colon \text{Int}, f(\text{Int}) : \text{Int} \\
D' & \colon \text{itsD1} \colon \text{Int}, f(\text{Int}) : \text{Int} \\
E' & \colon \text{itsC1} \colon \text{Bool}, f(\text{Float}) : \text{Int} \\
F' & \colon \text{itsD2} \colon \text{Float}, f(\text{Float}) : \text{Int}
\end{align*}
\]

**Notions** (from category theory):

- **invariance**,
- **covariance**,
- **contravariance**.

We could call, e.g. a method, **sub-type preserving**, if and only if it

- accepts **more general** types as input (contravariant),
- provides a **more specialised** type as output (covariant).

This is a notion used by many programming languages — and easily type-checked.
Excursus: Late Binding of Behavioural Features

Late Binding

What transformer applies in what situation? (Early (compile time) binding.)

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

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>someC -&gt; f()</td>
<td>&quot;</td>
<td>&quot;</td>
<td>(d)</td>
</tr>
<tr>
<td>someD -&gt; f()</td>
<td>&quot;</td>
<td>&quot;</td>
<td>(e)</td>
</tr>
<tr>
<td>someC -&gt; f()</td>
<td>&quot;</td>
<td>D::f()</td>
<td>(f)</td>
</tr>
</tbody>
</table>
Late Binding in the Standard and in Prog. Lang.

- 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, 2007b, 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”

**Exercise**: What could be the rationale of the designers of C++?

**Note**: late binding typically applies only to methods, not to attributes. (But: getter/setter methods have been invented recently.)

*Back to the Main Track: “...tell the difference...” for UML*
With Only Early Binding...

- ...we’re done (if we realise it correctly in the framework).
- Then
  - if we’re calling method $f$ of an object $u$,
  - which is an instance of $D$ with $C \preceq D$
  - via a $C$-link,
  - then we (by definition) only see and change the $C$-part.
  - We cannot tell whether $u$ is a $C$ or an $D$ instance.

So we immediately also have behavioural/dynamic subtyping.

Difficult: Dynamic Subtyping

- $C::f$ and $D::f$ are type compatible, but $D$ is not necessarily a sub-type of $C$.

Examples: (C++)

```
int C::f(int) {  
  return 0;  
};  
```

vs.

```
int D::f(int) {  
  return 1;  
};  
```

```
int C::f(int) {  
  return (rand() % 2);  
};  
```

vs.

```
int D::f(int x) {  
  return (x % 2);  
};  
```
In the standard, Section 7.3.36, "Operation":

**Semantic Variation Points**

[...] When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations. [OMG, 2007a, 106]

- So, better: call a method **sub-type preserving**, if and only if it
  1. accepts more input values (contravariant),
  2. on the old values, has fewer behaviour (covariant).

**Note:** This (ii) is no longer a matter of simple type-checking!

- And not necessarily the end of the story:
  - One could, e.g. want to consider execution time.
  - Or, like [Fischer and Wehrheim, 2000], relax to “fewer observable behaviour”, thus admitting the sub-type to do more work on inputs.

**Note:** “testing” differences depends on the **granularity** of the semantics.

- **Related:** “has a weaker pre-condition,” (contravariant),
  - “has a stronger post-condition.” (covariant).

**Ensuring Sub-Typing for State Machines**

- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.

- But the state machine of a sub-class cannot be drawn from scratch.

- Instead, the state machine of a sub-class can only be obtained by applying actions from a restricted set to a copy of the original one. Roughly (cf. User Guide, p. 760, for details),
  - add things into (hierarchical) states,
  - add more states,
  - attach a transition to a different target (limited).

- They **ensure**, that the sub-class is a **behavioural sub-type** of the super class. (But method implementations can still destroy that property.)

- Technically, the idea is that (by late binding) only the state machine of the most specialised classes are running.

  By knowledge of the framework, the (code for) state machines of super-classes is still accessible — but using it is hardly a good idea...
References


