Object-Oriented Software Construction

Bertrand Meyer
Lecture 22:
Typing issues, covariance
Anchored declarations

class

    DRIVER

feature

    partner: like Current
        -- This driver’s alternate.

    share (other: like Current) is
        -- Choose other as alternate.
        require
            other /= Void
        do
            partner := other
        end

end
Client covariance (parallel hierarchies)

DIGRAM

**DRIVER**

**TRUCKER**

**HEAVY_TRUCKER**

**VEHICLE**

**TRUCK**

**HEAVY_TRUCK**

assigned
assign
More need for anchoring

class
    DRIVER
feature
    ... parter, share as before...

assigned: VEHICLE
    -- This driver’s assigned vehicle.

assign (v: VEHICLE) is
    -- Choose v as assigned vehicle.
    require
        v /= Void
    do
        assigned := v
    end
end
Anchored declarations (client)

class

\textit{DRIVER}

feature

\ldots parter, \textit{share} as before\ldots

\textit{assigned: VEHICLE}
\hspace{1cm} -- This driver’s assigned vehicle.

\textit{assign (v: \textbf{like assigned}) is}
\hspace{1cm} -- Choose \textit{v} as assigned vehicle.

\textit{require}
\hspace{1cm} \textit{v} /= \textit{Void}
\textit{do}
\hspace{1cm} \textit{assigned} := \textit{v}
\textit{end}

end
Polymorphism and the attachment rule

- In an assignment \( x := y \), or the corresponding argument passing, the base class of the type of \( y \) must be a descendant of the type of \( x \).
$d1$: DRIVER
$bd1$: BIKER
$td1$: TRUCKER
...

-- Create a “trucker” and a “biker” objects:
create $bd1$
create $td1$

$d1 := td1$

$d1$.share ($bd1$)
Polymorphism and client covariance

\[ d1: \text{DRIVER} \]
\[ bd1: \text{BIKER} \]
\[ td1: \text{TRUCKER} \]
\[ v1: \text{VEHICLE} \]
\[ truck1: \text{TRUCK} \]
...

-- Create a “trucker” and a “biker” objects:
\[ \text{create } bd1 \]
\[ \text{create } td1 \]
\[ d1 := td1 \]
\[ v1 := truck1 \]
\[ d1.assign(v1) \]
class RECTANGLE

inherit POLYGON

export \{NONE\} add_vertex end

feature

... 

invariant

vertex_count = 4

end
Polymorphism and descendant hiding

\[ p: POLYGON \]
\[ r: RECTANGLE \]

... 
\[ \text{create } r \]

...

\[ p := r \]

...

\[ p.add\_vertex (...) \]
Genericity makes things worse!

drivers: LIST [DRIVER]
td1: TRUCKER
bd1, bd2: BIKER
track1: TRUCK

... drivers.extend (td1) ...
... drivers.extend (bd1) ...
... drivers.item (i).share (bd2)
drivers.item(i).assign (truck1)
Solutions to the catcall problem

- Solution 1: Novariance
- Solution 2: dynamic checks
- Solution 3 (described in *Eiffel: The Language*): system ee-validity

Compute set of all potential types for each entity. This will for example disallow the combined presence of

\[ d1 := b1 \quad d1.share(t1) \]

because one of the possible types for \( d1 \) is \textit{BIKER}.

- Also useful for compiler optimization.

- Disadvantage: requires (at least in a straightforward implementation) access to the entire system. Errors not easy to explain to users.
Static typing is pessimistic

(Pascal conventions)

\[
\begin{align*}
n & : \text{INTEGER} \\
n & := 0.0 \\
n & := 1.0 \\
n & := 3.67 \\
n & := 3.67 - 3.67
\end{align*}
\]
An entity $x$ is polymorphic if it satisfies one of the following properties:

- It appears in an assignment $x := y$ where $y$ is of a different type or (recursively) polymorphic.
- It is a formal routine argument.
- It is an external function.
A routine is a **CAT** (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments.

A call is a **catcall** if some redefinition of the routine would make it invalid because of a change of export status or argument type.
Polymorphic catcalls

\[ d1: \text{DRIVER} \]
\[ b1: \text{BIKER} \]
\[ t1: \text{TRUCKER} \]
\[ \text{create } b1 \]
\[ \text{create } t1 \]
\[ d1 := t1 \]

...\[ d1\text{.share}(t1) \]

\[ p: \text{POLYGON} \]
\[ r: \text{RECTANGLE} \]
\[ \text{create } r \]
\[ p := r \]
\[ p\text{.add_vertex}(\ldots) \]
New type rule:

- Polymorphic catcalls are invalid.

Note: can be checked incrementally; will be flagged either when adding a call or when adding a redefinition.

Not implemented by any compiler.
Yet another solution (5)

- Declare as **cat** any routine that either:
  - Has a redeclared argument type in a descendant
  - Has an anchored (**like**) argument
  - Has a more restricted export status in a descendant
- Declare as **poly** any entity that can be polymorphic.
- Prohibit:
  - Call of a cat routine on a poly target
  - Polymorphic assignment to target that is not poly
- The problem: foresight! (Open-closed principle).
End of lecture 23