Object-Oriented Software Construction

Bertrand Meyer
Lecture 21:
Typing issues, covariance
The America Conjecture

(Pierre America at TOOLS Europe 1990)

- One can have at most two of the three properties of:
  - Static typing.
  - Substitutivity.
  - Covariance.

- Can we disprove the America conjecture?
Typing: A simple problem!

- The basic operation of object-oriented computation:

\[ x.f(arg) \]
The typing problem

- When, and how, do we know that:
  - There is a feature applicable to \( OBJ \) and corresponding to \( f \)?
  - \( arg \) is an acceptable argument for that feature?
Terminology

\( x \): Entity.

**OBJ**: The object attached to \( x \) at some time during execution.

\( f \): Routine (One of the two forms of feature; the other is attributes.)
Typing vs. binding

- What do we know about the feature to be called?
- Static typing:
  - At least one
- Dynamic binding:
  - The right one
- Example:
  - my_aircraft.lower_landing_gear
Inheritance and typing

- deferred
- + effected
- ++ redefined

AIRCRAFT

* PLANE

+ lower_landing_gear

* COPTER

A_320

B_747

B_737

B_747_400

BOEING

AIRBUS

lower_landing_gear++
Cost of correcting errors (Boehm)
Typing rules (all locally checkable)

- Declaration rule.
- Call rule.
- Attachment rule.

(For the precise formulations see *Eiffel: The Language.*)
• Every entity must be declared as being of a certain type.

• For example:

\[ x: \text{AIRCRAFT} \]
\[ n: \text{INTEGER} \]
\[ ba1: \text{BANK\_ACCOUNT} \]
Call rule

- If a class $C$ contains the call
  
  $x.f$

- there must be a feature of name $f$ in the base class of the type of $x$, and that feature must be available (exported) to $C$. 
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 base class of the type of $x$. 
For typing to be acceptable

- No exception to type ee-rules ("casts").
- Multiple inheritance
- Unconstrained genericity:
  \[
  \textbf{class} \ LIST \ [G] \ldots
  \]
- Constrained genericity:
  \[
  \textbf{class} \ VECTOR \ [G \rightarrow \ NUMERIC] \ldots
  \]
Assignment attempt:

\[ x \equiv y \]

Contracts, to clarify the semantics and include constraints other than typing (e.g. numerical ranges).

Covariance for routine arguments.

Anchored declarations (like \( x \)) to avoid endless redeclarations.
Multiple inheritance

DOCUMENT

MESSAGE

MAILABLEDocumento
A class hierarchy

**DRIVER**
partner: DRIVER
share (other: DRIVER)

**TRUCKER**
partner++
share++

**HEAVY_TRUCKER**
partner++
share++

**BIKER**
partner++
share++
class DRIVER

feature

    partner: DRIVER
    -- This driver’s alternate

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

...

end
A typical call

\[ d_1, d_2 : \text{DRIVER} \]

\[ \ldots \]

\[ d_1.\text{share}(d_2) \]
class TRUCKER

inherit DRIVER

redefine partner

end

feature

partner: TRUCKER

-- This driver’s alternate.

...

end
The need for covariance

class TRUCKER
inherit DRIVER

redefine partner, share

end

feature

partner: TRUCKER
    -- This driver’s alternate.

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

end
Direct covariance

**DRIVER**
- partner: DRIVER
- share (other: DRIVER)

**TRUCKER**
- partner++
- share++

**HEAVY_TRUCKER**
- partner++
- share++

**BIKER**
- partner++
- share++
Avoiding redefinition avalanche

- Anchored declarations:

```plaintext
class TRUCKER
inherit DRIVER

redefine partner, share
end

feature

partner: TRUCKER
  -- This driver’s alternate.

share (other: like partner) is
  require other /= Void
  do partner := other
end
```

---

Chair of Software Engineering
End of lecture 22