Object-Oriented Software Construction

Bertrand Meyer

Lecture 6: Genericity
Genericity

Unconstrained

\textbf{LIST} [G]
e.g. LIST [INTEGER], LIST [PERSON]

Constrained

\textbf{HASH\_TABLE} [G \rightarrow \text{HASHABLE}]
\textbf{VECTOR} [G \rightarrow \text{NUMERIC}]
Extending the basic notion of class
Extending the basic notion of class
**Genericity: Ensuring type safety**

How can we define consistent “container” data structures, e.g. list of accounts, list of points?

Dubious use of a container data structure:

```plaintext
    c : COMPANY
    a : PERSON
    companies : LIST ...
    people : LIST ...

    companies.extend (c)
    people.extend (a)

    c := people.last

    c.change_recommendation (Buy)
```
Possible approaches

- Wait until run time; if types don’t match, trigger a run-time failure. (Smalltalk)

- Cast to a universal type, such as “pointer to void” in C.

- Duplicate code, manually or with help of macro processor.

- Parameterize the class, giving an explicit name \( G \) to the type of container elements. This is the Eiffel approach.
A generic class

class LIST [G] feature
extend (x: G) is ...
last: G is ...
end

To use the class: obtain a generic derivation, e.g.

companies: LIST [COMPANY]
Conformance rule

- $B [U]$ conforms to $A [T]$ if and only if $B$ is a descendant of $A$ and $U$ conforms to $T$. 
Using generic derivations

companies: LIST [COMPANY]
people: LIST [PERSON]
c: COMPANY
p: PERSON
...

companies.extend (c)
people.extend (p)

c := companies.last
c. change_recommendation (Buy)
...

Chair of Software Engineering
OOSC - Summer Semester 2005
Genericity and static typing

Compiler will reject

\[ \text{people.extend (c)} \]
\[ \text{companies.extend (p)} \]

To define more flexible data structures (e.g. stack of figures): use inheritance, polymorphism and dynamic binding.
Typing in an O-O context

An object-oriented language is statically typed if and only if it is possible to write a “static checker” which, if it accepts a system, guarantees that at run time, for any execution of a feature call $x.f$, the object attached to $x$ (if any) will have at least one feature corresponding to $f$. 
class VECTOR [G] feature
  infix "+" (other: VECTOR [G]): VECTOR [G] is
    -- Sum of current vector and other
    require
      lower = other.lower
      upper = other.upper
    local
      a, b, c: G
    do
      ... See next ...
    end
    ... Other features ...
end
Adding two vectors

\[ u + v = w \]

\[ i \]

\[ a + b = c \]
Constrained genericity

Body of `infix "+":`

```plaintext
create Result.make (lower, upper)

from
  i := lower
until
  i > upper
loop
  a := item (i)
  b := other.item (i)
  c := a + b  -- Requires "+" operation on G!
  Result.put (c, i)
  i := i + 1
end
```
Declare class `VECTOR` as

```plaintext
class VECTOR [G -> NUMERIC] feature
  ... The rest as before ...
end
```

Class `NUMERIC` (from the Kernel Library) provides features `infix "+", infix "*"` and so on.
Improving the solution

Make \textit{VECTOR} itself a descendant of \textit{NUMERIC}, effecting the corresponding features:

\begin{verbatim}
class VECTOR [G -> NUMERIC] inherit NUMERIC

feature
  ... The rest as before, including \texttt{infix "+"} ...
end
\end{verbatim}

Then it is possible to define

\begin{verbatim}
ν : VECTOR [INTEGER]
nν : VECTOR [VECTOR [INTEGER]]
nνν : VECTOR [VECTOR [VECTOR [INTEGER]]]
\end{verbatim}
Using arrays:

\[
\begin{align*}
\mathit{a}: &\quad \text{ARRAY [REAL]} \\
\text{...} \quad \text{create } &\quad \mathit{a}.\text{make} \ (1, \ 300) \\
\mathit{a}.\text{put} &\quad (3.5, \ 25) \\
\mathit{x} &\quad := \mathit{a}.\text{item} \ (i) \\
\end{align*}
\]

-- Alternatively: \( \mathit{x} := \mathit{a} \ @ \ i \)

\[
\begin{align*}
\text{infix } &\quad "@" \\
\end{align*}
\]

Also: \( \text{ARRAY2 [G]} \) etc.
Class **ARRAY (1)**

class ARRAY [G] create
  make
feature
  lower, upper: INTEGER

  count: INTEGER

  make (min: INTEGER, max: INTEGER) is
    -- Allocate array with bounds min and max.
    do
      ...
    end

Class **ARRAY** (2)

```literate
item, infix "@" (i: INTEGER): G is
    -- Entry of index i
    require
        lower <= i
        i <= upper
    do ... end

put (v: G; i: INTEGER) is
    -- Assign the value of v to the entry of index i.
    require
        lower <= i
        i <= upper
    do ... end

invariant
    count = upper - lower + 1
end
```

*Chair of Software Engineering*
End of lecture 6