Object-Oriented Software Construction

Bertrand Meyer
Lecture 6: Genericity
Genericity

- Parameterized classes for static typing
- Examples: stacks, arrays
- Constrained genericity (preview)
Extending the basic notion of class

Abstraction

Type parameterization

LIST_OF_PEOPLE

LIST_OF_BOOKS

SET_OF_BOOKS

LIST_OF_JOURNALS

Specialization

Type parameterization

LINKED_LIST_OF_BOOKS
Genericity: Ensuring type safety

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

- Dubious use of a container data structure:

```java
a: ACCOUNT
p, q: POINT
point_stack: STACK ...
account_stack: STACK ...

point_stack.put (p)
account_stack.put (a)
q := point_stack.item
q.move (3.4, 1.2)
```
Possible approaches

- Ignore the problem until run-time. Possible “feature not available” errors. This is the dynamic typing approach of Smalltalk and Java.

- Use a pseudo-universal type, such as “pointer to void” or “pointer to character” in C, to represent $G$ in the class, and cast everything to that type.

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

- Parameterize the class, giving a name $G$ to the type of container elements; the features of the class may refer to type $G$. This is the Eiffel approach. (See also C++ templates.)
A generic class

class STACK [G]

feature

  put (x: G) is ...

  item: G is ...

end

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

  account_stack: STACK [ACCOUNT]
- \[ B \[ U \] \] conforms to \( A \[ T \] \) if and only if \( B \) is a descendant of \( A \) and \( U \) conforms to \( T \).
Using generic derivations

account_stack: STACK [ACCOUNT]
point_stack: STACK [POINT]
a: ACCOUNT
p, q, r: POINT
...
point_stack.put (p)
point_stack.put (q)

r := point_stack.item
r. move (3.0, -5.0)
account_stack.put (a)
...
Genericity and static typing

- Compiler will reject

  \[
  \text{point\_stack.put}(a) \\
  \text{account\_stack.put}(p)
  \]

- To define more flexible data structures (e.g. stack of figures): use inheritance, polymorphism and dynamic binding.
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 \( f \).
Constrained genericity

\textbf{class} \textit{VECTOR} [\textit{G}]

\textbf{feature}

\textbf{infix} "+" (\textit{other}: \textit{VECTOR} [\textit{G}]): \textit{VECTOR} [\textit{G}] \textbf{is}

\begin{itemize}
  \item \textit{lower} = \textit{other}.\textit{lower}
  \item \textit{upper} = \textit{other}.\textit{upper}
\end{itemize}

\textbf{local}

\begin{itemize}
  \item \textit{a}, \textit{b}, \textit{c}: \textit{G}
\end{itemize}

\textbf{do}

\begin{itemize}
  \item \ldots \textit{See next} \ldots
\end{itemize}

\textbf{end}

\begin{itemize}
  \item \ldots \textit{Other features} \ldots
\end{itemize}

\textbf{end}
Constrained genericity (cont’d)

- The 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 a “+” operation on G!
  Result.put (c, i)
  i := i + 1
end
```
The solution

- Declare class \texttt{VECTOR} as

  \begin{verbatim}
  class VECTOR [G -> NUMERIC]
  feature
  ... The rest as before ...
  end
  \end{verbatim}

- Class \texttt{NUMERIC} (from the Kernel Library) provides features \texttt{infix} ",+", \texttt{infix} "*" and so on.

- Better yet: make \texttt{VECTOR} itself a descendant of \texttt{NUMERIC}, effecting the corresponding features:
Improving the solution

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

  \begin{verbatim}
  class VECTOR [G \rightarrow NUMERIC]
  inherit NUMERIC
  feature
    ... The rest as before, including infix "+" ...
  end
  \end{verbatim}

- Then it is possible to define e.g.

  \begin{verbatim}
  v: VECTOR [VECTOR [VECTOR [INTEGER]]]
  \end{verbatim}
A generic library class: Arrays

- Using arrays in a client

\[ a: \text{ARRAY} \left[ \text{REAL} \right] \]

...  
\text{create} \ a.\text{make} \ (1, \ 300)  

\text{a.put} \ (3.5, \ 25)  
\quad -- \ (\text{in Pascal: } a[25] := 3.5)  

\text{x := a.item} \ (i)  
\quad -- \ (\text{in Pascal: } x := a[i])  
\quad -- \ \text{Alternatively: } x := a @ i  
\quad -- \ \text{Using the function infix "@"}

- Also: \text{ARRAY2} \left[ G \right] \text{ etc.}
The ARRAY class

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
The ARRAY class (cont’d)

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
Obsolete features and classes

- How to reconcile progress with the protection of the installed base?

- Obsolete features and classes support smooth evolution.

- In class `ARRAY`:

```java
enter (i: V; v: T) is
  obsolete "Use `put (value, index)`"
  do
    put (v, i)
  end
```
class ARRAY_LIST [G]

obsolete

"[ Use MULTI_ARRAY_LIST instead (same semantics, but new name ensures more consistent terminology). Caution: do not confuse with ARRAYED_LIST (lists implemented by one array each). ]"

inherit MULTI_ARRAY_LIST [G]

end
End of lecture 7