Object-Oriented Software Construction
Bertrand Meyer

Lecture 6: Genericity

Genericity

Unconstrained
LIST [G]
  e.g. LIST [INTEGER], LIST [PERSON]

Constrained
HASH_TABLE [G  HASHABLE]
VECTOR [G  NUMERIC]
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:

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

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

c := people.last

   c.change_recommendation (Buy)
```

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]
```

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.

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

```java
... 
companies.extend (c) 
people.extend (p) 
... 
c := companies.last 
c. change_recommendation (Buy) 
... 
```

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 $f$.

Genericity and static typing

Compiler will reject

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

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

Constrained genericity

```java
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

\[ \mathbf{u} + \mathbf{v} = \mathbf{w} \]

\[ a + b = c \]

Constrained genericity

Body of infix "+":

create \texttt{Result.make} (lower, upper)

from

\[ i := \text{lower} \]

until

\[ i > \text{upper} \]

loop

\[ a := \text{item} (i) \]

\[ b := \text{other.item} (i) \]

\[ c := a + b \quad \text{-- Requires \texttt{"+"} operation on \texttt{G!}} \]

\texttt{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.

Improving the solution

Make \texttt{VECTOR} itself a descendant of \texttt{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}
  \texttt{v : VECTOR ['INTERGER']}
  \texttt{vv : VECTOR [VECTOR ['INTERGER']]} \\
  \texttt{v\texttt{v} : VECTOR [VECTOR [VECTOR ['INTERGER']]]}
\end{verbatim}
A generic library class: Arrays

Using arrays:

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

...  
\[
\text{create } a.\text{make}(1, 300)
\]
\[
a.\text{put}(3.5, 25)
\]
\[
x := a.\text{item}(i)
\]
-- Alternatively: \( x := a \_ \_ \_ \_ \text{item}(i) \)
-- Using the function
\[
infix "@"
\]

Also: \text{ARRAY2}[G]\] etc.

Class ARRAY (1)

\[
class \text{ARRAY}[G] \text{ create}
\]
\[
\text{make}
\]
\[
\text{feature}
\]
\[
\text{lower, upper: INTEGER}
\]
\[
\text{count: INTEGER}
\]
\[
\text{make}(\text{min: INTEGER, max: INTEGER})\]
-- Allocate array with bounds \( \text{min} \) and \( \text{max} \).
\[
do
\]
...  
\[
end
\]

Class ARRAY (2)

\[
\text{item, infix } "@"(i: \text{INTEGER}): G \text{ is}
\]
-- Entry of index \( i \)
\[
\text{require}
\]
\[
\text{lower} \leq i
\]
\[
i \leq \text{upper}
\]
\[
do...end
\]
\[
\text{put}(v: G; i: \text{INTEGER})\]
-- Assign the value of \( v \) to the entry of index
\[
i,
\]
\[
\text{require}
\]
\[
\text{lower} \leq i
\]
\[
i \leq \text{upper}
\]
\[
do...end
\]

\[
\text{invariant}
\]
\[
\text{count} = \text{upper} - \text{lower} + 1
\]
\[
end
\]

End of lecture 6