Software Architecture

Bertrand Meyer
Lecture 8: Genericity, Inheritance
Genericity

- Parameterized classes for static typing
- Examples: stacks, arrays
- Constrained genericity (preview)
Genericity

Unconstrained

LIST $[G]$

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

Constrained

HASH_TABLE $[G \rightarrow HASHABLE]$

VECTOR $[G \rightarrow NUMERIC]$
Extending the basic notion of class

- Abstraction
- Inheritance
- Genericity
- Specialization

- LIST_OF_STOCKS
- SET_OF_STOCKS
- LIST_OF_COMPANIES
- LINKED_LIST_OF_STOCKS
- LIST_OF_PERSONS

Type parameterization
Extending the basic notion of class

Inheritance

Genericity
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 \textit{LIST}[G] \ feature
   extend (x: G) is ...
   last: G is ...
end

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

\textit{companies}: \textit{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)`
...

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 \( f \).
Constrained genericity

class \texttt{VECTOR}[G] \texttt{feature}

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

\begin{quote}
\hspace{1em} -- Sum of current vector and other \\
\hspace{1em} \texttt{require}
\hspace{2em} lower = other.lower \\
\hspace{2em} upper = other.upper
\end{quote}

\texttt{local}

\hspace{1em} a, b, c: G

\texttt{do}

\hspace{2em} \ldots \texttt{See next} \ldots

\texttt{end}

\hspace{1em} \ldots \texttt{Other features} \ldots

\texttt{end}
Adding two vectors

\[ u + v = w \]

\[ i \]

\[ a + b = c \]
Constrained genericity

Body of infix "+":

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
The solution

Declare class \texttt{VECTOR} as

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

Class \texttt{NUMERIC} (from the Kernel Library) provides features \texttt{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}
\textbf{v} : VECTOR [INTEGER]
\textbf{vv} : VECTOR [VECTOR [INTEGER]]
\textbf{vvv} : VECTOR [VECTOR [VECTOR [INTEGER]]]
\end{verbatim}
A generic library class: Arrays

Using arrays:

\[ a : ARRAY[REAL] \]

\[ \ldots \]

\[ \text{create } a\text{.make}(1, 300) \]
\[ a\text{.put}(3.5, 25) \]

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

-- Alternatively: \[ x := a @ i \]
-- Using the function infix "@"

Also: \[ ARRAY2[G] \] 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)**

```plaintext
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
```
Complementary material

- OOSC2:
  - Chapter 10: Genericity
What is inheritance?

- Describe a new class as extension or specialization of an existing class. (With MULTIPLE inheritance it can be an extension of several existing classes.)

  - From the module viewpoint: if $B$ inherits from $A$, all the services of $A$ are potentially available in $B$ (possibly with a different implementation).

  - From the type viewpoint: inheritance is the “is-plus-but-except” relation. If $B$ inherits from $A$, whenever an instance of $A$ is required, an instance of $B$ will be acceptable.
What is inheritance?

**Class A feature**

```ruby
a is
do
-- Some Code
end
end
```

**Class B feature**

```ruby
b is
do
-- Some Code
end
end
```

**Class C**

```ruby
inhibit
A
```

**Class D**

```ruby
inhibit
A, B
```

**Feature**

```ruby
c is do
-- Some code
end
```

```ruby
a is do
-- Some code
end
```

```ruby
b is do
-- Some code
end
```

Inheritance statement

Features you inherit from parent(s)
Terminology

- Parent, Child
- Ancestor, Descendant
  - The ancestors of $B$ are $B$ itself and the ancestors of its parents.
- Proper ancestor, Proper descendant
- Direct instance, Instance
  - The instances of $A$ are the direct instances of its descendants.
- (Other terminology: subclass, superclass, base class)
A deferred class has at least one deferred feature.

A deferred feature is a feature that has no feature body but only the feature declaration with its signature.
An effective class is a class that inherits from a deferred class and implement at least one of the deferred features of the deferred class.
Terminology: Redefinition of a feature of a parent class

class C
feature f is
  do
    -- Some other code
  end
end

class D
inherit C
  redefine f
end
feature f is
  do
    -- Some other code
  end
end

C and D are normal classes – they are not deferred and not effective. Here they are used to show how a feature \( f \) from class \( C \) can be redefined in class \( D \).
Example: Inheritance hierarchy
Example: POLYGON

class POLYGON
create make
feature
  vertices: ARRAY [POINT]
  vertices_count: INTEGER

perimeter: REAL is
  -- Perimeter length
  do
    from ... until ... loop
      Result := Result + (vertices @ i) . distance (vertices @ (i + 1))
  end
end

invariant
  vertices_count >= 3
  vertices_count = vertices.count
end
Example: RECTANGLE by redefining POLYGON

class
  RECTANGLE

inherit
  POLYGON

redefine perimeter end

create
  make

feature
  diagonal, side1, side2: REAL
  perimeter: REAL is
    -- Perimeter length
    do
      Result := 2 * (side1 + side2)
    end

invariant
  vertices_count = 4

end
End of lecture 8