Software Architecture

Bertrand Meyer

Lecture 8: Genericity, Inheritance

Genericity

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

Extending the basic notion of class

Abstract

Inheritance

Type parameterization

Genericity

Specialization

LIST_OF_STOCKS

SET_OF_STOCKS

LIST_OF_COMPANIES

LINKED_LIST_OF_STOCKS

LIST_OF_PERSONS

Extending the basic notion of class

Inheritance

SET_OF_STOCKS

SET_OF_COMPANIES

LIST_OF_STOCKS

LINKED_LIST_OF_COMPANIES

LIST_OF_PERSONS

LINKED_LIST_OF_COMPANIES

Constrained

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:

\[ c : \text{COMPANY} \]
\[ a : \text{PERSON} \]
\[ \text{companies}: \text{LIST} \ldots \]
\[ \text{people}: \text{LIST} \ldots \]
\[ \text{companies}.\text{extend}(c) \]
\[ \text{people}.\text{extend}(a) \]
\[ c := \text{people}.\text{last} \]
\[ c.\text{change_recommendation}(\text{Buy}) \]

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

\[ \text{class } \text{LIST}[G] \]
\[ \text{feature} \]
\[ \text{extend}(x : G) \text{ is } \ldots \]
\[ \text{last} : G \text{ is } \ldots \]
\[ \text{end} \]

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

\[ \text{companies: } \text{LIST}[\text{COMPANY}] \]

Using generic derivations

\[ \text{companies: } \text{LIST}[\text{COMPANY}] \]
\[ \text{people: } \text{LIST}[\text{PERSON}] \]
\[ c : \text{COMPANY} \]
\[ p : \text{PERSON} \]
\[ \ldots \]
\[ \text{companies}.\text{extend}(c) \]
\[ \text{people}.\text{extend}(p) \]
\[ c := \text{companies}.\text{last} \]
\[ c.\text{change_recommendation}(\text{Buy}) \]

Conformance rule

\[ B[U] \text{ conforms to } A[T] \text{ if and only if } B \text{ is a descendant of } A \text{ and } U \text{ conforms to } T. \]

Genericity and static typing

Compiler will reject

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

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 VECTOR \([G \rightarrow \textbf{NUMERIC}]\) feature

infix "+"(other VECTOR \([G]\]): VECTOR \([G]\)

\(-\) 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

\[ U + V = W \]

\[ \begin{array}{ccc}
 i & a & + & b & = & c \\
 \hline
 \end{array} \]

Constrained genericity

Body of infix "+":

create 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 "+" operation on \(G\)!}\)
Result.put (c, i)
\(i := i + 1\)
end

The solution

Declare class VECTOR as

class VECTOR \([G \rightarrow \textbf{NUMERIC}]\) feature

... The rest as before ...
end

Class NUMERIC (from the Kernel Library) provides features infix "+", infix ";" and so on.

Improving the solution

Make VECTOR itself a descendant of NUMERIC, effecting the corresponding features:

class VECTOR \([G \rightarrow \textbf{NUMERIC}]\) inherit

NUMERIC

feature

... The rest as before, including infix "+"...
end

Then it is possible to define

\[ v: \text{VECTOR [INTEGER]} \]
\[ vv: \text{VECTOR [VECTOR [INTEGER]]} \]
\[ vvv: \text{VECTOR [VECTOR [VECTOR [INTEGER]]]} \]
A generic library class: Arrays

Using arrays:

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

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

\[ a \text{.put}(3.5, 25) \]

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

-- Alternatively: \[ x := a @ i \]

-- Using the function infix "@"

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

Class ARRAY (1)

\[ \text{class ARRAY}[, G] \text{.create} \]

\[ \text{make} \]

\[ \text{feature} \]

\[ \text{lower, upper: INTEGER} \]

\[ \text{count: INTEGER} \]

\[ \text{make}(\text{min, INTEGER, max, INTEGER}) \text{is} \]

\[ \text{do} \]

-- Allocate array with bounds min and max.

\[ \text{end} \]

Class ARRAY (2)

\[ \text{item, infix "@" (i: INTEGER): G} \text{is} \]

-- Entry of index \( i \)

\[ \text{require} \]

\[ \text{lower} \leq i \leq \text{upper} \]

\[ \text{do} \text{...} \text{end} \]

\[ \text{put(v: G; i: INTEGER)} \text{is} \]

-- Assign the value of \( v \) to the entry of index \( i \)

\[ \text{require} \]

\[ \text{lower} \leq i \leq \text{upper} \]

\[ \text{do} \text{...} \text{end} \]

\[ \text{invariant} \]

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

end

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.
### 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)

### Terminology: Deferred classes

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.

### Terminology: Effective classes and features

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

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

- **POLYGON**
  - create
  - make
  - feature
    - vertices: ARRAY [POINT] vertices_count: INTEGER
    - perimeter: REAL
    - from ... until ... loop
    - Result := Result + (vertices @ i).distance (vertices @ (i + 1))
  - end
  - Invariant
    - vertices_count >= 3
    - vertices_count = vertices.count
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