Programming in the large

Bertrand Meyer
Lecture 8: Genericity, Inheritance
Agenda for today

- Genericity
- Inheritance
  - Terminology
  - Example
Agenda for today

- Genericity
  - Inheritance
    - Terminology
    - Example
Genericity

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

- Abstraction
  - SET_OF_BOOKS
  - LIST_OF_PEOPLE
  - LINKED_LIST_OF_BOOKS

- Type parameterization
  - LIST_OF_BOOKS
  - LIST_OF_JOURNALS

- Specialization
Genericity: Ensuring type safety

- How can we define consistent “container” data structures, e.g. stack of accounts, stack of points?
- Consistency should be ascertained at compile time
Type-safe use

\[ a : \text{ACCOUNT} \; ; \; p, q : \text{POINT} \]

\[ \text{point\_stack} : \text{STACK} \; \ldots \]
\[ \text{account\_stack} : \text{STACK} \; \ldots \]

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

\[ q : = \text{point\_stack}\text{.item} \]

\[ q\text{.move} \; (3.4, 1.2) \]
Type-unsafe use

*a: ACCOUNT ; p, q: POINT*

*point_stack: STACK ...*

*account_stack: STACK ...*

*point_stack.put (p)*

*account_stack.put (a)*

*q := account_stack.item*

*q.move (3.4, 1.2)*
Possible approaches

- Write specific classes: `STACK_OF_ACCOUNTS`, `STACK_OF_POINTS` etc. Code duplication
- Ignore until run-time: possible errors if feature not available (see Smalltalk, current Java).
- Use pseudo-universal type, such as “pointer to void” in C, to represent $G$; cast everything to it.
- Make the class explicitly generic, as in Eiffel (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]
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

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

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

An O-O language is **statically typed** if and only if it is possible to write a tool (static checker) which, if it accepts a program, 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 for \(f\).

Alternative strategy: **dynamic** typing (check at run time)

- Smalltalk
- Many non-O-O languages, e.g. Lisp, Perl, Haskell…
About static typing

- One solution: reject all programs!
- The issue is to find the right balance between safety and flexibility
- Example with integer variable \( n \):

\[
\text{if } \text{False} \text{ then}
\]
\[
\quad n := 1.0
\]
\[
\text{end}
\]
Using arrays in a client

\[ a : ARRAY [REAL] \]

\[ a.make (1, 300) \]

\[ a.put (3.5, 25) \]

--- (in Pascal: \( a[25] := 3.5 \))

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

--- (in Pascal: \( x := a[i] \))

--- Alternatively: \( x := a @ i \)

--- Using the function infix "@"

Also: \( ARRAY2 [G] \) etc.
class ARRAY [G]

create
make

feature
lower, upper: INTEGER

count: INTEGER

make (min: INTEGER, max: INTEGER) is
  -- Allocate array with bounds min and max.
  require
    max >= min - 1
  do
    ...
  ensure
    lower = min
    upper = max
end
item, infix "@" (i: INTEGER): G is
  -- Entry of index i
require
  lower <= i
  i <= upper
do
  ...
end

put (v: G; i: INTEGER) is
  -- Set entry of index i to v.
require
  lower <= i
  i <= upper
do
  ...
end
invariant
  count = upper – lower + 1
end
Complementary material

- OOSC2:
  - Chapter 10: Genericity
Agenda for today

- Genericity
- Inheritance
  - Terminology
  - Example
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
  a is
do
  -- Some Code
end
end

class B feature
  b is
do
  -- Some Code
end
end

Inheritance statement

Features you inherit from parent(s)

class A feature
  a is
do
  -- Some Code
end
end

Inheritance statement

Features you inherit from parent(s)

class C
  inherit
    A

feature
c is
do
  -- Some code
end

class D
  inherit
    A, B

feature
c is
do
  -- Some code
end

a is
do
  -- Some code
end
end

b is
do
  -- Some code
end
end
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.
Terminology: Effective classes and features

**class**

\[
\text{class } B \\
\text{inherit } A \\
\text{feature } f \text{ is } \\
\text{do } \\
\quad \text{-- Some code} \\
\text{end} \\
\text{end}
\]

* deferred

+ effective

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
dend
feature f is
do
  -- Some other code
end
end`

`++ redefined`

\[ C \]
\[ D \]

\[ f \]

\[ f ++ \]

\( C \) 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

- FIGURE
  - OPEN_FIGURE
    - SEGMENT
    - POLYLINE
  - CLOSED_FIGURE
    - POLYGON
      - TRIANGLE
      - SQUARE
    - ELLIPSE
      - CIRCLE

Perimeter:
- deferred
- effective
- redefined

Extent:
- center
- rotate
- display
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