Programming in the large

Bertrand Meyer

Lecture 8: Genericity, Inheritance

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

Type parameterization

LIST OF PEOPLE

LIST OF BOOKS

SET 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

```plaintext
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)
```

Type-unsafe use

```plaintext
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

```plaintext
class STACK [G]
  feature
    put (x: G) is ...
    item: G is ...
  end
```

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

```plaintext
account_stack: STACK [ACCOUNT]
```

Using generic derivations

```plaintext
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
  
  \[
  \text{point_stack}.\text{put} \ (a) \\
  \text{account_stack}.\text{put} \ (p)
  \]

- 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 \( n \):
  
  \[
  \text{if} \ \text{False} \ \text{then} \\
  n := 1.0 \\
  \text{end}
  \]

A generic library class: Arrays

- Using arrays in a client
  
  \[
  a: \text{ARRAY} \ [\text{REAL}] \\
  \ldots \\
  \text{create} \ a.\text{make} \ (1, \ 300) \\
  a.\text{put} \ (3.5, \ 25) \\
  \quad \quad \quad \quad \quad \quad \text{-- (in Pascal: a[25] := 3.5)} \\
  x := a.\text{item} \ () \\
  \quad \quad \quad \quad \quad \quad \text{-- (in Pascal: x := a[i])} \\
  \quad \quad \quad \quad \quad \quad \text{-- Alternatively: x := a[i]} \\
  \quad \quad \quad \quad \quad \quad \text{-- Using the function infix "@"}
  \]

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

The ARRAY class

\[
\text{class} \ \text{ARRAY} \ [\ G] \\
\text{create} \\
\text{make} \\
\text{feature} \\
\quad \text{lower, upper: INTEGER} \\
\quad \text{count: INTEGER} \\
\quad \text{make (min: INTEGER, max: INTEGER) is} \\
\quad \quad \text{-- Allocate array with bounds min and max.} \\
\quad \quad \text{require} \\
\quad \quad \quad \text{max} \ >\ = \ \text{min} \ - \ 1 \\
\quad \quad \text{do} \\
\quad \quad \quad \text{...} \\
\quad \quad \text{ensure} \\
\quad \quad \quad \text{lower} = \ \text{min} \\
\quad \quad \quad \text{upper} = \ \text{max} \\
\quad \text{end}
\]

\[
\text{item, infix "@" (i: INTEGER): G is} \\
\quad \text{require} \\
\quad \quad \text{Entry of index i} \\
\quad \quad \text{lower} \ <= \ i \\
\quad \quad \text{i} \ <= \ \text{upper} \\
\quad \quad \text{do} \\
\quad \quad \quad \text{...} \\
\quad \quad \text{end} \\
\quad \text{put (v: G; i: INTEGER) is} \\
\quad \quad \text{require} \\
\quad \quad \quad \text{Set entry of index i to v.} \\
\quad \quad \quad \text{lower} \ <= \ i \\
\quad \quad \quad \text{i} \ <= \ \text{upper} \\
\quad \quad \quad \text{do} \\
\quad \quad \quad \quad \text{...} \\
\quad \quad \text{end} \\
\quad \text{invariant} \\
\quad \quad \text{count} = \ \text{upper} - \ \text{lower} + 1 \\
\quad \text{end}
\]
Complementary material

- OOSC2:
  - Chapter 10: Genericty

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.

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.

Example: Inheritance hierarchy

Example: POLYGON

Example: RECTANGLE by redefining POLYGON

End of lecture 8