Programmer 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
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

```java
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

```java
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]

Using generic derivations

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
  
  ```
  point_stack.put(a)
  account_stack.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 \( n \):
  
  ```
  if False then
    n := 1.0
  end
  ```
A generic library class: Arrays

- Using arrays in a client
  
  \[ a: \text{ARRAY} \left[ \text{REAL} \right] \]
  
  `create a.make \left( 1, 300 \right)`
  
  `a.put \left( 3.5, 25 \right)`
  
  `x := a.item \left( i \right)`
  
  **Also:** `ARRAY2 \left[ \text{G} \right]` etc.

The ARRAY class

\[
\begin{align*}
\text{class} & \quad \text{ARRAY} \left[ \text{G} \right] \\
\text{create} & \quad \text{make} \\
\text{feature} & \quad \text{lower}, \text{upper: INTEGER} \\
\text{count: INTEGER} & \\
\text{make} \left( \text{min: INTEGER}, \text{max: INTEGER} \right) & \quad \text{allocate array with bounds min and max.} \\
\text{require} & \quad \text{max} \geq \text{min} - 1 \\
\text{do} & \quad \text{...} \\
\text{ensure} & \quad \text{lower} = \text{min} \\
& \quad \text{upper} = \text{max} \\
\end{align*}
\]

\[
\begin{align*}
\text{item, infix \text{"@"} \left( \text{i: INTEGER} \right): \text{G} \text{ is} \\
\text{require} & \quad \text{lower} \leq \text{i} \\
\text{do} & \quad \text{upper} \leq \text{i} \\
\text{end} & \\
\end{align*}
\]

\[
\begin{align*}
\text{put} \left( v: \text{G}; \text{i: INTEGER} \right) & \quad \text{set entry of index i to v.} \\
\text{require} & \quad \text{lower} \leq \text{i} \\
\text{do} & \quad \text{lower} \leq \text{i} \\
\text{end} & \\
\text{invariant} & \quad \text{count} = \text{upper} - \text{lower} + 1 \\
\end{align*}
\]
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
  - Some Code

- Class B
  - Feature b
  - Some Code

**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)

**Terminology: Deferred classes**

- **deferred class** A
  - Feature f is deferred
    - No code
  - end

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 implements 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
Example: POLYGON

```java
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
      invariant vertices_count >= 3
      vertices_count = vertices.count
    end
  end
```

Example: RECTANGLE by redefining POLYGON

```java
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
```

End of lecture 8