Introduction to Programming

Bertrand Meyer

Lecture 13:
Inheritance & genericity

Last revised 11 December 2005

Inheritance basics

Principle:
Describe a new class as extension or specialization of an existing class (or several with multiple inheritance)

If $B$ inherits from $A$:

- Module viewpoint: all the services of $A$ are available in $B$ (possibly with different implementation).
- Type viewpoint: whenever an instance of $A$ is required, an instance of $B$ will be acceptable ("is-a" relationship)

Terminology

Parent, Heir

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)
Example hierarchy

Redefinition 1: polygons

```plaintext
class POLYGON inherit
  CLOSED_FIGURE
create
  make
  feature
    vertex: ARRAY[POINT] INTEGER
    vertex_count: REAL
  perimeter: REAL is -- Perimeter length
  do from ... until ... loop
    Result = Result + vertex[i].distance(vertex[i+1])
  end
end
```

Redefinition 2: rectangles

```plaintext
class RECTANGLE inherit
  POLYGON
  redefine
  perimeter
create
  make
  feature
    diagonal, side1, side2: REAL
  perimeter: REAL is -- Perimeter length
  do Result = 2 * (side1 + side2) end
end
```
Inheritance, typing and polymorphism.

- Assume:
  - \( p \) : POLYGON, \( r \) : RECTANGLE, \( t \) : TRIANGLE,
  - \( x \) : REAL

- Permitted:
  - \( x := p . \text{perimeter} \)
  - \( x := r . \text{perimeter} \)
  - \( x := r . \text{diagonal} \)
  - \( p := r \)

- NOT permitted:
  - \( x := p . \text{diagonal} \) (even after \( p := r \))
  - \( r := p \)

Dynamic binding

- What is the effect of the following (assuming \texttt{some\_test} true)?
  ```
  if \texttt{some\_test} then
    \( p := r \)
  else
    \( p := t \)
  end
  \( x := p . \text{perimeter} \)
  ```

- Redefinition: A class may change an inherited feature, as with POLYGON redefining perimeter.
- Polymorphism: \( p \) may have different forms at run-time.
- Dynamic binding: Effect of \( p . \text{perimeter} \) depends on run-time form of \( p \).

Without inheritance!

```
\texttt{display (f, \text{FIGURE})} is
  do
    if \texttt{"f is a CIRCLE"} then
      ...
    elseif \texttt{"f is a POLYGON"} then
      end
      ...
    end
  end
```

and similarly for all other routines!

Tedious; must be changed whenever there's a new figure type.
With inheritance!

With:

\[
\begin{align*}
&\text{f.}\ \text{FIGURE} \\
&\text{c.}\ \text{CIRCLE} \\
&\text{p.}\ \text{POLYGON}
\end{align*}
\]

and:

\[
\begin{align*}
&\text{create}\ \text{c.make}\ (...) \\
&\text{create}\ \text{p.make}\ (...) \\
\end{align*}
\]

Initialize:

\[
\begin{align*}
&\text{if} \ ... \ \text{then} \\
&\quad \text{f} := c \\
&\text{else} \\
&\quad \text{f} := p \\
&\text{end}
\end{align*}
\]

Then just use:

\[
\begin{align*}
&\text{f.move}\ (...) \\
&\text{f.rotate}\ (...) \\
&\text{f.display}\ (...) \\
&\quad \text{-- and so on for every} \\
&\quad \text{operation on f!}
\end{align*}
\]

Inheritance: summary 1

- Type mechanism: lets you organize our data abstractions into taxonomies
- Module mechanism: lets you build new classes as extensions of existing ones
- Polymorphism: Flexibility \textit{with} type safety
- Dynamic binding: automatic adaptation of operation to target, for more modular software architectures

Genericity

Unconstrained

\[
\begin{align*}
&\text{LIST}[G] \\
&\quad \text{e.g. LIST [INTEGER], LIST [PERSON]}
\end{align*}
\]

Constrained

\[
\begin{align*}
&\text{HASH_TABLE}[G \rightarrow \text{HASHABLE}] \\
&\text{VECTOR}[G \rightarrow \text{NUMERIC}]
\end{align*}
\]
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:

```python
    c: CITY; p: PERSON
cities: LIST...
    people: LIST...
    people.extend(p)
cities.extend(c)
    c := cities.last
    c. add_tram_line(LineB)
```

What if wrong?
### Possible approaches

1. Duplicate code, manually or with help of macro processor.
2. Wait until run time; if types don’t match, trigger a run-time failure. (Smalltalk)
3. Convert (“cast”) all values to a universal type, such as “pointer to void” in C.
4. Parameterize the class, giving an explicit name $G$ to the type of container elements. This is the Eiffel approach, now being adopted also by Java, .NET and others.

### A generic class

```plaintext
class LIST[G] feature
  extend(x: G) is ... 
  last: G is ...
end

To use the class: obtain a generic derivation, e.g.
  cities: LIST[CITY]

Actual generic parameter
```

### Using generic derivations

```plaintext
cities: LIST[CITY]
people: LIST[PERSON]
c: CITY
p: PERSON
...
  cities.extend(c)
  people.extend(p)

c := cities.last

STATIC TYPING:
The compiler will reject:
  - people.extend(c)
  - cities.extend(p)
```

```plaintext
  c.add_tram_line(Line8)
```
### Static typing

**Type-safe call (during execution):**
A feature call \( x.f \) such that the object attached to \( x \) has a feature corresponding to \( f \).

*(Generalizes to calls with arguments, \( x.f(a, b) \))*

**Static type checker:**
A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be **type-safe**.

**Statically typed language:**
A programming language for which it is possible to write a static type checker.

---

### The static and the dynamic

For a feature call \( x.f \):

- **Static typing:**
  There is at least one feature \( f \) applicable to \( x \)

- **Dynamic binding:**
  If more than one possible feature, execution will select the right feature

---

### Using genericity

- \( \text{LIST [CITY]} \)
- \( \text{LIST [LIST [CITY]]} \)
- ...

A type is no longer exactly the same thing as a class!

(But every type remains based on a class.)
Genericity: summary 1

- Type extension mechanism
- Reconciles flexibility with type safety
- Enables us to have parameterized classes
- Useful for container data structures: lists, arrays, trees, ...
- "Type" now a bit more general than "class"

Genericity + inheritance 1: Constrained genericity

```plaintext
class VECTOR[G] feature
  plus alias "+" (other VECTOR[G]): VECTOR[G] is
  -- 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

\[ \begin{array}{ccc}
U & + & V \\
\hline
2 & 1 & 0 \\
\hline
\end{array} \]

\[ \begin{array}{ccc}
2 & + & 1 \\
\hline
\color{red}a & + & \color{yellow}b \\
\hline
\color{red}c \\
\end{array} \]
Constrained genericity

Body of plus alias "+":

```plaintext
create Result.make (lower, upper)
from
  /= lower
until
  /= upper
loop
  a := item ()
  b := other.item ()
  c := a + b 
     -- Requires "+" operation on G!
  Result.put (c, i)
end
```

The solution

Declare class VECTOR as

```plaintext
class VECTOR (G -> NUMERIC) feature
  ... The rest as before ...
end
```

Class NUMERIC (from the Kernel Library) provides features plus alias "+", minus alias "-" and so on.

Improving the solution

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

```plaintext
class VECTOR (G -> NUMERIC) inherit
  NUMERIC
  feature
    ... Rest as before, including infix "+" ...
  end
```

Then it is possible to define

```plaintext
v : VECTOR [INTEGER]
vv : VECTOR [VECTOR [INTEGER]]
vvv : VECTOR [VECTOR [VECTOR [INTEGER]]]
```
Extending the basic notion of class

Inheritance

LIST OF CARS

SET OF CARS

SET OF PERSONS

LIST OF CITIES

LIST OF CARS

LIST OF PERSONS

LINKED LIST OF CITIES

LINKED LIST OF CARS

Genericity

Genericity + Inheritance 2: Polymorphic data structure

class LIST[k]
feature
  last k is ...
  extend(x; k) is ...
end

fl. LIST[FIGURE]
fl. SQUARE
fl. TRIANGLE
fl. POLYGON
...
fl. extend(p); fl. extend(f); fl. extend(s); fl. extend(r)
fl. last, display

Example hierarchy

FIGURE

OPEN FIGURE

CLOSURE FIGURE

SEGMENT

POLYLINE

POLYGON

ELLIPSE

TRIANGLE

RECTANGLE

SQUARE

CIRCLE

deferred

effective

redefined
Forcing a type: the problem

```plaintext
f1.store("FILE_NAME")
...
-- Two years later:
    f1:= retrieved("FILE_NAME")
    x := fl.last -- [1]
    print (x.diagonal) -- [2]

But:
  ➢ If x is declared of type RECTANGLE, [1] is invalid.
  ➢ If x is declared of type FIGURE, [2] is invalid.
```

The solution: Assignment attempt

```plaintext
f1.store("FILE_NAME")
...
-- Two years later:
    f1?: retrieved("FILE_NAME")
    x := fl.last -- [1]
    print (x.diagonal) -- [2]

But:
  ➢ If x is declared of type RECTANGLE, [1] is invalid.
  ➢ If x is declared of type FIGURE, [2] is invalid.
```

Assignment attempt

```plaintext
f: FIGURE
r: RECTANGLE
...
fl.retrieve("FILE_NAME")
    f := fl.last
        r? := f
    if r /= Void then
        print (r.diagonal)
    else
        print("Too bad")
end
```
Assignment attempt

\[ x := y \]

with \[ x : A \]

- If \( y \) is attached to an object whose type conforms to \( A \), perform normal reference assignment.
- Otherwise, make \( x \) void.

Inheritance and assertions

Correct call:

\[
\text{if } a1.a \text{ then }
\]

\[
\text{else }
\]

\[
\text{end}
\]

Assertion redeclaration rule

- Redefined version may not have require or ensure.
- May have nothing (assertions kept by default), or

\[
\text{require else new_pre ensure then new_post}
\]

- Resulting assertions are:
  - original_precondition or new_pre
  - original_postcondition and new_post
Invariant accumulation

- Every class inherits all the invariant clauses of its parents.
- These clauses are conceptually "and"-ed.

End of lecture 13