Introduction to Programming

Bertrand Meyer

Last revised 15 December 2003
Lecture 17: Inheritance
What is inheritance?

- Principle: Describe a new class not from scratch but as extension or specialization of one existing class — or several in the case of MULTIPLE inheritance.

  - 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, 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

FIGURE

OPEN_FIGURE

SEGMENT

POLYLINE

POLYGON

CLOSED_FIGURE

ELLIPSE

CIRCLE

TRIANGLE

RECTANGLE

SQUARE

extent*
barycenter*
... display*
rotate...
perimeter*
perimeter+
diagonal
perimeter++
side1
side2
perimeter++
perimeter++
... deferred
+ effective
++ redefined
Redefinition 1: polygons

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
Redefinition 2: rectangles

class RECTANGLE  inherit

POLYGON

redefine

perimeter

end

create

make

feature

diagonal, side1, side2: REAL

perimeter: REAL is

-- Perimeter length

do

Result := 2 * (side1 + side2)

do

end

invariant

vertices_count = 4

end
Assume:

\[ p: \text{POLYGON}; \ r: \text{RECTANGLE}; \ t: \text{TRIANGLE}; \]
\[ x: \text{REAL} \]

Permitted:

\[ x := p.\text{perimeter} \]
\[ x := r.\text{perimeter} \]
\[ x := r.\text{diagonal} \]
\[ p := r \]

NOT permitted:

\[ x := p.\text{diagonal} \] (even just after \( p := r \) !)
\[ r := p \]
Dynamic binding

- What is the effect of the following (assuming `some_test` true)?

```plaintext
if some_test then
    p := r
else
    p := t
end
x := p.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.perimeter` depends on run-time form of `p`. 
Non-O-O techniques

\[
\text{display } (f: \text{FIGURE}) \text{ is}
\]
\[
\quad \text{do}
\]
\[
\quad \text{if } "f \text{ is a CIRCLE}" \text{ then}
\]
\[
\quad \quad \text{...}
\]
\[
\quad \text{elseif } "f \text{ is a POLYGON}" \text{ then}
\]
\[
\quad \quad \text{...}
\]
\[
\quad \text{end}
\]
\[
\text{end}
\]

and similarly for all other routines!
With inheritance techniques

\[ f: \text{FIGURE} \]
\[ c: \text{CIRCLE} \]
\[ p: \text{POLYGON} \]

... 
create \( c.\text{make} (...) \)
create \( p.\text{make} (...) \)

... 
if ... then
  \( f := c \)
else
  \( f := p \)
end

...
\( f.\text{move} (...) \)
\( f.\text{rotate} (...) \)
\( f.\text{display} (...) \)
...
class STACK [G]
feature
  ...
  item: G is ...
  put (x: G) is ...
end

fs: STACK [FIGURE]

r: RECTANGLE
s: SQUARE
t: TRIANGLE
p: POLYGON
...
fs.put (p); fs.put (t); fs.put (s); fs.put (r)
fs.item.display
Example hierarchy

* deferred
+ effective
++ redefined

FIGURE

OPEN FIGURE

SEGMENT POLYLINE

POLYGON

CLOSED FIGURE

POLYGON

TRIANGLE

RECTANGLE

SQUARE

ELLIPSE

CIRCLE

extent* barycenter* ...

display* rotate *

perimeter* perimeter+ diagonal perimeter++ ...

perimeter+ perimeter+ perimeter++ side1 side2

perimeter++ perimeter++
Genericity vs. Inheritance

- **Type parameterization**
  - `LIST_OF_PEOPLE`
  - `LIST_OF_BOOKS`
  - `LIKED_LIST_OF_BOOKS`
  - `SET_OF_BOOKS`

- **Abstraction**
  - `LIST_OF_BOOKS`

- **Specialization**
  - `LIST_OF_JOURNALS`
Forcing a type: the problem

\texttt{fs.store} ("FILE\_NAME")

\texttt{...}

\texttt{-- Two years later:}

\[
\texttt{fs := retrieved ("FILE\_NAME")}
\]

\[
\texttt{x := fs.item \quad -- [1]}
\]

\[
\texttt{print (x.diagonal) \quad -- [2]}
\]

But:

- If \texttt{x} is declared of type \texttt{RECTANGLE}, [1] is invalid.
- If \texttt{x} is declared of type \texttt{FIGURE}, [2] is invalid.
The solution: Assignment attempt

\[ fs\text{.store} \text{("FILE\_NAME")} \]

...  

-- Two years later:  
\[
\begin{align*}
fs \ &= \ \text{retrieved} \text{("FILE\_NAME")} \\
x \ &= \ fs\text{.item} \quad \text{-- [1]} \\
\text{print} \ (x\text{.diagonal}) \quad \text{-- [2]}
\end{align*}
\]

But:
- If \( x \) is declared of type \( \text{RECTANGLE} \), [1] is invalid.
- If \( x \) is declared of type \( \text{FIGURE} \), [2] is invalid.
Assignment attempt

\[ f: \text{FIGURE} \]
\[ r: \text{RECTANGLE} \]

\[
... \\
fs.\text{retrieve} \("\text{FILE\_NAME}\") \\
f := \text{fs.item} \\
\text{if } r \neq \text{Void} \text{ then} \\
\quad \text{print} \(r.\text{diagonal}\) \\
\text{else} \\
\quad \text{print} \("\text{Too bad.}\") \\
\text{end}
\]
Assignment attempt (cont’d)

\[ x \ ?= \ y \]

with

\[ x: A \]

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

feature

infix "+" (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
Constrained genericity

- The body of **infix** "+":

```plaintext
create Result.make (lower, upper) from
  i := lower until
  i > upper loop
  a := item (i)
  b := other.item (i)
  c := a + b -- Requires a "+" operation on G!
  Result.put (c, i)
  i := i + 1
end
```
Constrained genericity

- The body of **infix** "+":

```plaintext
create Result.make (lower, upper)
from
  i := lower
until
  i > upper
loop
  a := item (i)
  b := other.item (i)
  c := a + b  -- Requires a "+" operation on G!
  Result.put (c, i)
  i := i + 1
end
```
The solution

- Declare class `VECTOR` as

  ```
  class VECTOR [G -> NUMERIC]
  feature
    ... The rest as before ...
  end
  ```

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

- Better yet: make `VECTOR` itself a descendant of `NUMERIC`, effecting the corresponding features:
Improving the solution

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

  ```
  class VECTOR [G -> NUMERIC]
  inherit
  NUMERIC
  feature
  ... The rest as before, including infix "+"...
  end
  ```

- Then it is possible to define e.g.

  ```
  v: VECTOR [VECTOR [VECTOR [INTEGER]]]
  ```
Inheritance and assertions

Correct call:

```
if a1.α then
  a1.r (...)
else
  ...
end
```
Assertion redeclaration rule

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

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

- Resulting assertions are:
  - \text{original_precondition or new_pre}
  - \text{original_postcondition and new_post}
Invariant accumulation

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