



Object-Oriented Software Construction

Bertrand Meyer

Lecture 7: Inheritance



Agenda for today

- Inheritance
 - Example
 - Polymorphism and dynamic binding
- Genericity
 - Assignment attempt

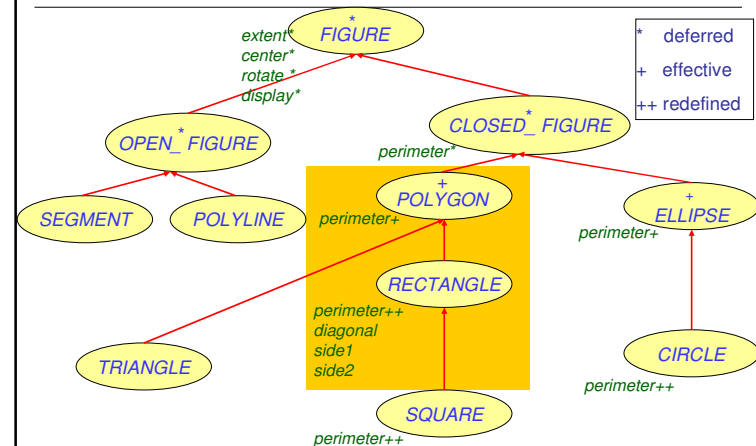


Agenda for today

- Inheritance
 - Example
 - Polymorphism and dynamic binding
- Genericity
 - Assignment attempt



Example: Inheritance hierarchy



Example: POLYGON

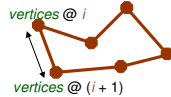
5

```

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

```



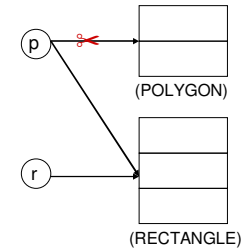
Polymorphism and dynamic binding

7

- Assume:
 - p : POLYGON; r : RECTANGLE; t : TRIANGLE;
 - x : REAL

- Permitted:
 - $x := p.perimeter$
 - $x := r.perimeter$
 - $x := r.diagonal$
 - $p := r$

- NOT permitted:
 - $x := p.diagonal$ (even just after $p := r$!)
 - $r := p$



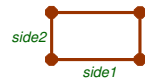
Example: RECTANGLE by redefining POLYGON

6

```

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

```



Polymorphism and dynamic binding

8

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

```

if some_test then
  p := r
else
  p := t
end
x := p.perimeter

```

- Redefinition:** A class may change an inherited feature, as with *RECTANGLE* redefining perimeter of *POLYGON*.
- Polymorphism:** p may have different forms at run-time.
- Dynamic binding:** Effect of $p.perimeter$ depends on run-time form of p .

Dynamic binding: Using non-O-O techniques

9

```

display (f: FIGURE) is
do
  if "f is a CIRCLE" then
    ...
  elseif "f is a POLYGON" then
    ...
  end
end
    
```

and similarly for all other routines!

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

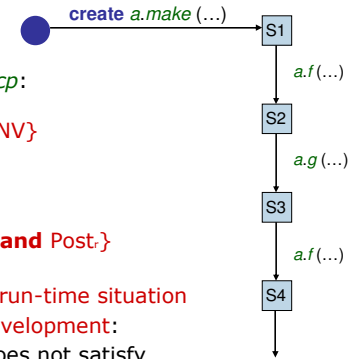
The dangers of static binding

11

- For every creation procedure cp :

$$\{Pre_{cp}\} \mathbf{do}_{cp} \{Post_{cp} \mathbf{and} INV\}$$
- For every exported routine r :

$$\{INV \mathbf{and} Pre_r\} \mathbf{do}_r \{INV \mathbf{and} Post_r\}$$
- The worst possible erroneous run-time situation in object-oriented software development:
 - Producing an object that does not satisfy the invariant of its class.



Dynamic binding: in action

10

With:

```

figure_list: LIST [FIGURE]
c: CIRCLE
p: POLYGON
f: FIGURE
    
```

and:

```

create c.make
create p.make
create figure_list.make
    
```

Initialize:

```

figure_list.extend (c)
figure_list.extend (p)
    
```

Then just use:

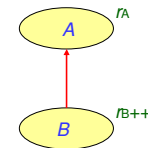
```

f := figure_list.i_th (i)
f.move (...)
f.rotate (...)
f.display
-- and so on for every
-- operation on f
    
```

The dangers of static binding

12

- $\{INV_A\} \mathbf{do}_{r_A} \{INV_A\}$
- $\{INV_B\} \mathbf{do}_{r_B} \{INV_B\}$
- Consider a call of the form $a1.r$ where $a1$ is polymorphic:
 - No guarantee on the effect of \mathbf{do}_{r_A} on an instance of B !



A concrete example

13

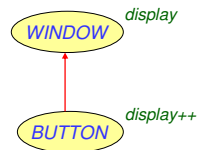
w: WINDOW

b: BUTTON

create *b*

w := *b*

w.display



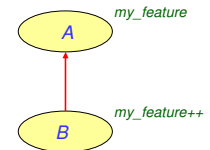
Use of Precursor

15

May have arguments.

```

class
  B
inherit
  A
feature
  redefine my_feature end
  my_feature (args: SOME_TYPE) is
  do
    -- Something here
    Precursor {A} (args)
    -- Something else here
  end
end
  
```



Using original version of redefined feature

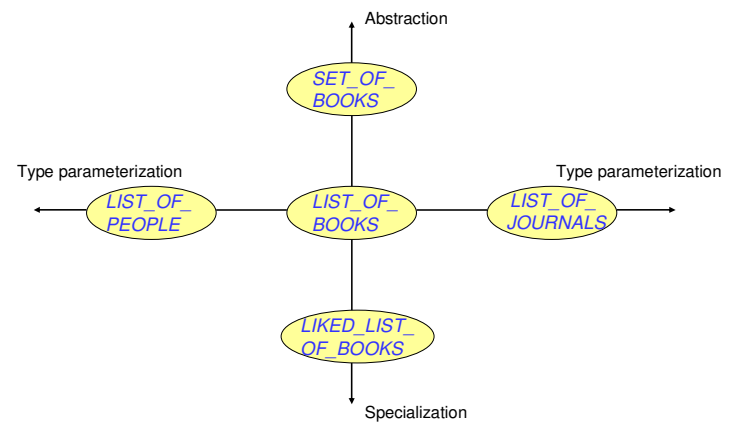
14

```

class
  BUTTOn
inherit
  WINDOW
  redefine display end
feature
  display is
  do
    Precursor {WINDOW}
    display_border
    display_label
  end
  display_label is do ... end
  display_border is do ... end
end
  
```

Genericity vs. Inheritance

16



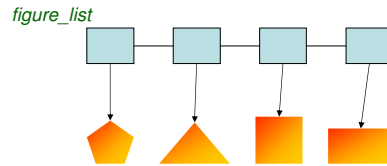
Genericity: *LIST* [G]

17

```

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

figure_list: LIST [FIGURE]
r: RECTANGLE
s: SQUARE
t: TRIANGLE
p: POLYGON
figure_list.extend (p)
figure_list.extend (t)
figure_list.extend (s)
figure_list.extend (r)
figure_list.last.display
  
```



Genericity: Forcing a type - the problem

19

```

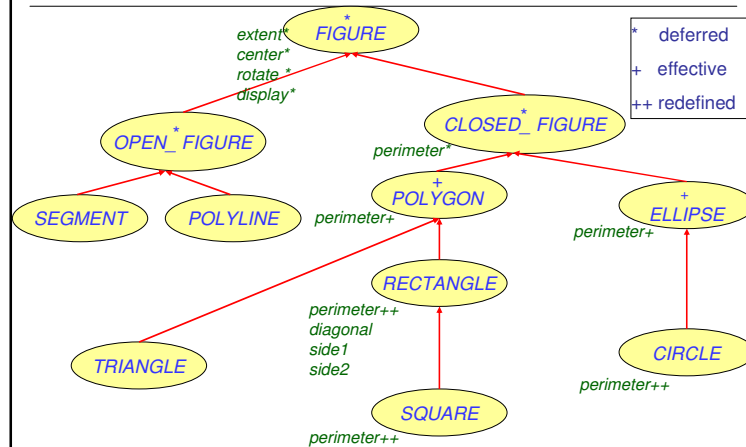
figure_list.store ("FILE_NAME")
...
-- Two years later:
figure_list := retrieved ("FILE_NAME")
x := figure_list.last -- [1]
print (x.diagonal) -- [2]
  
```

But:

- If x is declared of type *RECTANGLE*, [1] is invalid.
- If x is declared of type *FIGURE*, [2] is invalid.

Example: Inheritance hierarchy

18



The solution: Assignment attempt

20

$x ?= y$

with

$x: A$

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



Forcing a type: The solution (using an assignment attempt)

21

```
f: FIGURE
r: RECTANGLE
...
figure_list := retrieved ("FILE_NAME")
f := figure_list.last
r ?= f
if r /= Void then
    print (r.diagonal)
else
    print ("Too bad.")
end
```



22

End of lecture 7