Object-Oriented Software Construction
Bertrand Meyer

Lecture 7: Inheritance

Agenda for today

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

Example: Inheritance hierarchy

[Diagram showing inheritance hierarchy]
Example: POLYGON

```plaintext
class POLYGON
create make
feature
  vertices: ARRAY [POINT]
  perimeter: REAL is
    do from ... until ... loop
      Result := Result + (vertices @ i) . distance (vertices @ (i + 1))
    end
  invariant
    vertices_count >= 3
    vertices_count = vertices.count
end
```

Polymorphism and dynamic binding

- 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 \( p := r \) !)
  \( r := p \)

Example: RECTANGLE by redefining POLYGON

```plaintext
class RECTANGLE
inherit POLYGON
  redefine perimeter end
create make
feature
  diagonal, side1, side2: REAL
  perimeter: REAL is
    do
      Result := 2 * (side1 + side2)
    end
  invariant
    vertices_count = 4
end
```

Polymorphism and dynamic binding

- What is the effect of the following (assuming `some_test` is true)?
  ```plaintext
  if some_test then
    p := r
  else
    p := t
  end
  x := p\text{.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 \( p\text{.perimeter} \) depends on run-time form of \( p \).
Dynamic binding: Using non-O-O techniques

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

and similarly for all other routines!

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

Dynamic binding: in action

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 (1)
f.move (...) f.rotate (...)
f.display
  -- and so on for every
  -- operation on f
```
A concrete example

- `w`: WINDOW
- `b`: BUTTON
- `create b`
- `w := b`
- `w.display`

Use of Precursor

May have arguments.

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

Using original version of redefined feature

- `class BUTTON`
- `inhibit 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

- `Abstraction` vs. `Specialization`
- `Type parameterization`
Genericity: \textit{LIST} \([G]\)

\begin{verbatim}
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
\end{verbatim}

Genericity: Forcing a type - the problem

\begin{verbatim}
figure_list.store ("FILE_NAME")
...
end

-- Two years later:
figure_list := retrieved ("FILE_NAME")
x := figure_list.last   -- [1]
print (x.diagonal)   -- [2]
\end{verbatim}

But:
\begin{itemize}
  \item If \(x\) is declared of type \textit{RECTANGLE}, [1] is invalid.
  \item If \(x\) is declared of type \textit{FIGURE}, [2] is invalid.
\end{itemize}

Example: Inheritance hierarchy

\begin{verbatim}
x != y
\end{verbatim}

with
\begin{verbatim}
x: A
\end{verbatim}

\begin{itemize}
  \item If \(y\) is attached to an object whose type conforms to \(A\), perform normal reference assignment.
  \item Otherwise, make \(x\) void.
\end{itemize}
Forcing a type: The solution (using an assignment attempt)

```
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
```

End of lecture 7