Object-Oriented Software Construction
Bertrand Meyer

Lecture 8: More on inheritance

Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance

Adding two vectors

\[
\begin{align*}
U &+ V = W \\
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array}
\end{align*}
\]
Constrained genericity

```plaintext
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
```

Adding two vectors

```
\[ U + V = W \]
```

```
\[ i \]
```

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 solution

- Declare class VECTOR as
  ```plaintext
  class VECTOR [G -> NUMERIC]

  feature
    ... The rest as before ...
  end
  ```

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

Creating with a specified type

- To avoid this:
  
  ```
  a1: A
  b1: B
  ...
  create b1.make (...)
  a1 := b1
  ```

- Simply use
  
  ```
  a1: A
  ...
  create {B} a1.make (...)
  ```

(See factory pattern)

Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance

Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance
Once routines

- If instead of
  \[
  r \text{ is } \begin{cases} \text{do} & \text{... Instructions ...} \\ \text{end} \end{cases}
  \]
  you write
  \[
  r \text{ is } \begin{cases} \text{once} \text{ ... Instructions ...} \\ \text{end} \end{cases}
  \]
- then Instructions will be executed only for the first call by any client during execution. Subsequent calls return immediately.
- In the case of a function, subsequent calls return the result computed by the first call.

Agenda for today

- Constrained genericity
- Creating with a specified type
- Once routines
- Multiple inheritance

Scheme for shared objects

```plaintext
class SHARED_OBJECTS
  feature
  error_window: WINDOW is
    once create Result.make (...)
  end
  exit_button: BUTTON is
    once create Result.make (...)
  end
end

class My_APPLICATION_CLASS inherit
  SHARED_OBJECT
feature
  r is
    do
      error_window.put (my_error_message)
    end
end
```

Multiple inheritance

- Allow a class to have two or more parents.
- Examples that come to mind: ASSISTANT inherits from TEACHER and STUDENT.
**Example: Teaching assistant**

- This is in fact a case of **repeated** inheritance:

  ![Diagram of repeated inheritance](image1.png)

**Multiple inheritance: Combining abstractions**

- COMPARABLE
- NUMERIC
- INTEGER
- REAL
- STRING
- DOUBLE
- COMPLEX

**Other examples of multiple inheritance**

- Combining separate abstractions:
  - Restaurant, train car
  - Calculator, watch
  - Plane, asset

**Multiple inheritance: Composite figures**

- Simple figures
- A composite figure
Defining the notion of composite figure

A composite figure as a list

Composite figures through multiple inheritance

Composite figures

class COMPOSITE Figure
inherit
  FIGURE redefine display, move, rotate, ... end
  LIST[FIGURE]
feature
  display is -- Display each constituent figure in turn.
  do 
  from start until after loop
  item.display forth
  end
end
... Similarly for move, rotate etc. ...
end
**Complex figures**

- A simpler form of procedures *display, move* etc. can be obtained through the use of iterators.
- We’ll learn to use *agents* for that purpose.

**Multiple inheritance from interfaces: limitations**

- It is often useful to have a mix of abstract and concrete ("effective") features
- Eiffel "deferred" classes permit this.
- Not possible in Java and the .NET object model
- Java experience shows that programmers resort to various ugly tricks to simulate this... (See John Viega, TOOLS USA 2000)

---

**Multiple inheritance**

---

**deferred class** COMPARABLE [G] feature

```
infixed"<"(other: COMPARABLE [G]): BOOLEAN is deferred

infixed"<="(other: COMPARABLE [G]): BOOLEAN is
do
Result := Current < other or
equal (Current, other)
end

infixed">="(other: COMPARABLE [G]) is ...
infixed">"(other: COMPARABLE [G]) is ...
...
end -- class COMPARABLE
```
Multiple inheritance: Name clashes

Resolving name clashes

class C
inherit A

rename foo as fog
end

rename foo as zoo
end

feature ...

Resolving name clashes

Results of renaming

a1: A
b1: B
c1: C
...
c1.fog
c1.zoo
a1.foo
b1.foo

Invalid:
a1.fog, a1.zoo, b1.zoo, b1.fog, c1.foo
When is a name clash acceptable?

- (Between n features of a class, all with the same name, immediate or inherited.)
  - They must all have compatible signatures.
  - If more than one is effective, they must all come from a common ancestor feature under repeated inheritance.

Another application of renaming

- Provide locally better adapted terminology.
- Example: *child* (*TREE*); *subwindow* (*WINDOW*).

Feature merging

```
class D inherit
  A
  B
  C
feature
  ...
end
```
class D inherit
  A
    rename  g as f
    end
  B
  C
    rename  h as f
    end
feature ...
end

a1: A  b1: B  c1: C  d1: D
a1.g  b1.f  c1.h  d1.f
End of lecture 8