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
Agenda for today

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

\[ u + v = w \]

\[
\begin{array}{ccc}
  & a & b & c \\
  \hline \\
  i & & & \\
  \hline \\
  & & & \\
\end{array}
\]
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
```

Chair of Software Engineering  OOSC - Summer Semester 2005
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
```
Adding two vectors

\[ u + v = w \]
Constrained genericity: 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.
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.

  ```
  ν: VECTOR [VECTOR [VECTOR [INTEGER]]]
  ```
Agenda for today

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

- To avoid this:

  \[
  \begin{align*}
  a1 &: A \\
  b1 &: B \\
  \ldots \\
  \text{create } & b1.\text{make } (\ldots) \\
  a1 &: b1 \\
  \end{align*}
  \]

- Simply use

  \[
  \begin{align*}
  a1 &: A \\
  \ldots \\
  \text{create } & \{B\} a1.\text{make } (\ldots) \\
  \end{align*}
  \]

(See factory pattern)
Agenda for today

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

- If instead of
  
  \[
  r \text{ is} \\
  \text{do} \\
  \text{end}
  \]
  
  \[
  \ldots \text{Instructions} \ldots
  \]

- you write

  \[
  r \text{ is} \\
  \underline{\text{once}} \\
  \text{do} \\
  \underline{\text{end}}
  \]
  
  \[
  \ldots \text{Instructions} \ldots
  \]

- then \textit{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.
class SHARED_OBJECTS
feature
  error_window: WINDOW is
    once
      create Result.make (...)
    end
end

exit_button: BUTTON is
  once
    create Result.make (...)
  end
end

class MY_APPLICATION_CLASS
inherit
  SHARED_OBJECTS
feature
  r is
    do
      error_window.put (my_error_message)
    end
end
Agenda for today

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

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

![Diagram](image-url)
Example: Teaching assistant

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

```
  UNIVERSITY_MEMBER
     ^
   /   \
  ??   ??
 |     |
|     |
|     |
TEACHER ?????
    |
    |
    id

        UNIVERSITY_MEMBER
     id
     /   \
   ??   ??
  

            UNIVERSITY_MEMBER
     id
     /   \
   ??   ??
        

            UNIVERSITY_MEMBER
     id
     /   \
   ??   ??

            UNIVERSITY_MEMBER
     id
     /   \
   ??   ??

```

Chair of Software Engineering

OOSC - Summer Semester 2005
Other examples of multiple inheritance

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

- COMPARABLE
- NUMERIC
- INTEGER
- REAL
- STRING
- DOUBLE
- COMPLEX
Multiple inheritance: Composite figures

Simple figures

A composite figure
Defining the notion of composite figure

![Diagram showing the relationships between FIGURE, LIST, and COMPOSITE FIGURE with actions such as display, hide, rotate, move, count, put, remove.

FIGURE

LIST

[FIGURE]

COMPOSITE FIGURE

display
hide
rotate
move
count
put
remove
Composite figures through multiple inheritance

- **FIGURE**
- **LIST[FIGURE]**
  - **COMPOSITE FIGURE**
    - **OPEN FIGURE**
    - **CLOSED FIGURE**
      - **SEGMENT**
      - **POLYLINE**
      - **POLYGON**
        - **TRIANGLE**
        - **RECTANGLE**
          - **CIRCLE**
            - **SQUARE**

**Properties:**
- perimeter
- perimeter*
- perimeter+
- perimeter**
- diagonal
- perimeter++
A composite figure as a list

start

item

forth

after
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

- COMPARABLE
- NUMERIC
- STRING
- DOUBLE
- INTEGER
- REAL
- MATRIX

*Comparable* and *Numeric* are superclasses, with *String*, *Double*, and *Matrix* as subclasses.
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)
deferred class COMPARABLE [G] feature

infix "<" (other: COMPARABLE [G]): BOOLEAN is deferred
end

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

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

- Rename `foo` as `fog` in `C`.
- Rename `foo` as `zoo` in `C`.

Diagram:

- Node A with `foo` incoming edge.
- Node B with `foo` incoming edge.
- Node C.
- Edge from A to C with label "rename foo as fog".
- Edge from B to C with label "rename foo as zoo".
Resolving name clashes

```ruby
class C
  inherit A

  rename foo as fog
  end

rename foo as zoo
end

feature ...
```
Results of renaming

\[\begin{align*}
a1 & : A \\
b1 & : B \\
c1 & : C \\
... & \\
c1.\text{fog} & \\
c1.\text{zoo} & \\
a1.\text{foo} & \\
b1.\text{foo} & \\
\end{align*}\]

Invalid:
\[a1.\text{fog}, a1.\text{zoo}, b1.\text{zoo}, b1.\text{fog}, c1.\text{foo}\]
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 (cont’d)

class $D$ inherit

  $A$

  $B$

  $C$

feature

  ...

end
Feature merging: with different names

A

B

C

D

g

f

h


Chair of Software Engineering
OOSC - Summer Semester 2005
class D inherit

A

rename g as f
end

B

C

rename h as f
end

feature
...
end
Feature merging: effective features

\[ g^+ \rightarrow A \quad f^+ \rightarrow B \quad h^+ \rightarrow C \]

\[ g \cap f \quad f^- \quad h \cap f \quad f^- \]

\[ a1: A \quad b1: B \quad c1: C \quad d1: D \]

\[ a1.g \quad b1.f \quad c1.h \quad d1.f \]
Feature merging: effective features

class D inherit
  A
    rename g as f
    undefine f
    end
  B
  C
    rename h as f
    undefine f
    end
feature
  ...
end
End of lecture 8