Einführung in die Programmierung

Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 10
Today

- Multiple inheritance
Inheritance is never the only way

Given the classes

- TRAIN_CAR, RESTAURANT

how would you implement a DINER?

- You could have an attribute in TRAIN_CAR
  train_service: SERVICE

- Then have RESTAURANT inherit from SERVICE

- This is flexible if the kind of service may change to a type that is unrelated to TRAIN_CAR

- Changes in TRAIN_CAR do not affect SERVICE easily
Examples of multiple inheritance

Combining separate abstractions:

- Restaurant, train car
- Calculator, watch

- Other examples?
  - Teacher, student
  - Home, vehicle
Multiple inheritance: Combining abstractions

- **COMPARABLE**
  - <, <=, >, >=, ...
  - (total order relation)
- **NUMERIC**
  - +, –, *, /, ...
  - (commutative ring)
- **INTEGER**
- **REAL**
- **STRING**
- **COMPLEX**
Composite figures
Multiple inheritance: Composite figures

Simple figures

A composite figure
Defining the notion of composite figure

center
display
hide
rotate
move
...

FIGURE

V_LIST
[FIGURE]

COMPOSITE_ FIGURE

count
put
remove
...

...
In the overall structure

- FIGURE
  - CLOSED FIGURE
    - SEGMENT
    - POLYLINE
    - POLYGON
    - TRIANGLE
  - OPEN FIGURE
    - POLYLINE
    - POLYGON
    - RECTANGLE
    - CIRCLE
    - SQUARE
  - ELLIPSE
- COMPOSITE FIGURE
  - V_LIST [FIGURE]

Perimeter calculations:
- perimeter
- perimeter*
- perimeter**
- diagonal

Note: The image contains a tree structure with nodes representing different types of figures and their relationships.
A composite figure as a list

before  

item  

forth  

Cursor  

after
Composite figures

class COMPOSITE FIGURE inherit FIGURE

V_LIST[FIGURE]

feature

display

-- Display each constituent figure in turn.
do

from start until after loop

item.display

end

... Similarly for move, rotate etc. ...

end

Requires dynamic binding
An alternative solution: the composite pattern

- **FIGURE**
  - **OPEN FIGURE**
  - **CLOSED FIGURE**
    - **SEGMENT**
    - **POLYLINE**
    - **POLYGON**
    - **ELLIPSE**
      - **RECTANGLE**
      - **CIRCLE**
        - **SQUARE**

- **LIST [FIGURE]**
  - **COMPOSITE FIGURE**

- Perimeter:
  - **perimeter**
  - **perimeter**
  - **perimeter**
  - **perimeter**
  - **diagonal**
  - **perimeter**
  - **perimeter**
  - **perimeter**
  - **perimeter**

- **figure_list**
The Java-C# solution

No multiple inheritance for classes

“Interfaces”: specification only (but no contracts)
  ➢ Similar to completely deferred classes (with no effective feature)

A class may inherit from:
  ➢ At most one class
  ➢ Any number of interfaces
Lessons from this example

Typical example of *program with holes*

We need the full spectrum from fully abstract (fully deferred) to fully implemented classes

Multiple inheritance is there to help us combine abstractions
Multiple inheritance: Name clashes

Hands-On
Resolving name clashes

\[ f \rightarrow A \]
\[ B \rightarrow f \]

rename \( f \) as \( A_f \)

\[ C \rightarrow A_f, f \]
Consequences of renaming

Valid or invalid?

\[ a_1: A \]
\[ b_1: B \]
\[ c_1: C \]
...

\[ c_1.f \] Valid
\[ a_1.A_f \] Invalid
\[ c_1.A_f \] Valid
\[ b_1.f \] Valid
\[ b_1.A_f \] Invalid

rename \( f \) as \( A_f \)

Hands-On
Are all name clashes bad?

A name clash must be removed unless it is:

- Under repeated inheritance (i.e. not a real clash)
- Between features of which at most one is effective (i.e. others are deferred)
Feature merging

\[ f^* \rightarrow A \quad f^* \rightarrow B \quad f^* \rightarrow C \quad f^+ \]

* Deferred
+ Effective
Feature merging: with different names

class D
inherit A
    rename g as f
end

B

C
    rename h as f
end

feature ...
end

Deferred
Effective
Renaming
Feature merging: effective features

- $f^+$
- $A$
- $B$
- $C$
- $D$

Deferred

Effective

Undefine
deferred class
  \( T \)
inherit
  \( S \)
  undefined \( v \) end

feature

  ...

end
Merging through undefined

class D
    inherit A
    undefined f end
    B
    C
    undefined f end
feature ...
end

* Deferred
+ Effective
-- Undefine
Merging effective features with different names

```
class D
  inherit A
    undefine f end
  B
    rename g as f
    undefine f end
  C
    rename h as f
    end
feature ...
end
```
Acceptable name clashes

If inherited features have all the same names, there is no harmful name clash if:

- They all have compatible signatures
- At most one of them is effective

Semantics of such a case:

- Merge all features into one
- If there is an effective feature, it imposes its implementation
Feature merging: effective features

\[ \begin{align*}
g^+ & \rightarrow A \\
f^+ & \rightarrow B \\
h^+ & \rightarrow C \\
 g \wedge f & \rightarrow D \\
h \wedge f & \rightarrow D
\end{align*} \]

\[
\begin{align*}
a1: & \quad A \\
b1: & \quad B \\
c1: & \quad C \\
d1: & \quad D \\
a1.g: & \quad b1.f \\
b1.f: & \quad c1.h \\
c1.h: & \quad d1.f
\end{align*}
\]
Exercise: All-in-one-device

Hands-On
Exercise: All-in-one-device

```plaintext
class PRINTER
    feature
        print_page -- Print a page.
            do
                print("Printer prints a page...")
            end
        switch_on -- Switch from 'off' to 'on'
            do
                print("Printer switched on...")
            end
    end

class FAX
    feature
        send -- Send a page over the phone net.
            do
                print("Fax sends a page...")
            end
        start -- Switch from 'off' to 'on'
            do
                print("Fax switched on...")
            end
    end

class SCANNER
    feature
        scan_page -- Scan a page.
            do
                print("Scanner scans a page...")
            end
        switch_on -- Switch from 'off' to 'on'
            do
                print("Scanner switched on...")
            end
        send -- Send data to PC.
            do
                print("Scanner sends data...")
            end
    end
```

Hands-On
Exercise: All-in-one-device

How to resolve the name clashes?

- switch_on
- send
Exercise: All-in-one-device

```plaintext
class ALL_IN_ONE_DEVICE

inherit Printer
    rename
        switch_on as start
    undefine
        start
    end

Scanner
    rename
        switch_on as start,
        send as send_data
    end

Fax
    rename
        send as send_message
    undefine
        start
    end

feature ... end
```
Valid or invalid?

```ruby
class ALL_IN_ONE_DEVICE
  inherit PRINTER
  rename
    switch_on as start
  undefine
    start
  end

  SCANNER
  rename
    switch_on as start,
    send as send_data
  end

  FAX
  rename
    send as send_message
  undefine
    start
  end

  feature ... end

s: SCANNER
f: FAX
a: ALL_IN_ONE_DEVICE

- a.switch_on: Invalid
- a.print_page: Valid
- f.send_message: Invalid
- s.switch_on: Valid
- f.send: Valid
- a.send: Invalid
```
A special case of multiple inheritance

This is a case of repeated inheritance
Indirect and direct repeated inheritance
Multiple is also repeated inheritance

A typical case:

```
copy++
is_equal++
```

```
D
```

```
LIST
```

```
C
```

```
ANY
```

```
copy
is_equal
```

```
copy
C_copy
is_equal
C_is_equal
```

??
Features such as $f$, not renamed along any of the inheritance paths, will be shared.

Features such as $g$, inherited under different names, will be replicated.
The need for select

A potential ambiguity arises because of polymorphism and dynamic binding:

\[ a1 : \text{ANY} \]
\[ d1 : D \]

... 

\[ a1 := d1 \]
\[ a1.copy(...) \]
Removing the ambiguity

class

\[ D \]

inherit

\[ V_{LIST}[T] \]

select

\[
\text{copy, is\_equal}
\]

end

\[ C \]

rename

\[
\text{copy as } C_{\_copy, is\_equal as } C_{\_is\_equal, ...
\]

end
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.