Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 11
Mock exam next week!

- Monday exercise groups: December 5
- Tuesday exercise groups: December 6
- You have to be present
- The week after we will discuss the results
Today

- Multiple inheritance
Combining abstractions

Given the classes

- TRAIN_CAR, RESTAURANT

how would you implement a DINER?
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?
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

LIST
[FIGURE]

COMPOSITE FIGURE

count
put
remove
...

...
In the overall structure

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

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

- **perimeter**
- **perimeter***
- **perimeter**
- **diagonal**
- **perimeter**
- **perimeter**
- **perimeter**
- **perimeter**
- **perimeter**
- **perimeter**
A composite figure as a list

before

item

forth

Cursor

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

Requires dynamic binding
An alternative solution: the composite pattern

COMPOSITE FIGURE

LIST
[FIGURE]

figure_list

OPEN FIGURE
CLOSED FIGURE

POLYLINE
POLYGON
ELLIPSE
SEGMENT
RECTANGLE
CIRCLE
SQUARE
TRIANGLE

perimeter*
perimeter*
diagonal
perimeter*
perimeter**
perimeter**
perimeter**

figure_list
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
Resolving name clashes

rename $f$ as $A_f$
Consequences of renaming

**Valid or invalid?**

\[\begin{align*}
a_1 & : A \\
b_1 & : B \\
c_1 & : C \\
\vdots \\
c_1.f & : \text{Valid} \\
a_1.A_f & : \text{Invalid} \\
c_1.A_f & : \text{Valid} \\
b_1.f & : \text{Valid} \\
b_1.A_f & : \text{Invalid} \\
\end{align*}\]
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

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

* 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

...
Feature merging: effective features

A

B

C

D

f^+

f^+

f^+

f^{--}

f^{--}

f^{--}

* Deferred
+ Effective
-- Undefine
deferred class
  T
inherit
  S
    undefine v end
feature
  ... 
end
Merging through undefined

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

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

```plaintext
class D
  inherit A
    undefine f end
  B
    rename g as f
    undefine f end
  C
    rename h as f
    undefine 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
Exercise: All-in-one-device

Hands-On

**Diagram:**

- **PRINTER**
- **SCANNER**
- **FAX**

**ALL_IN_ONE_DEVICE**
Exercise: All-in-one-device

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
class ALL_IN_ONEDEVICE
    inherit ...
end

How to resolve the name clashes?

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

A typical case:

```
copy ++
```

```
is_equal ++
```

```
LIST
```

```
ANY
```

```
C
```

```
D
```

```
??
```

```
copy
```

```
is_equal
```

```
C-copy
```

```
C_is_equal
```

Sharing and replication

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: ANY

\[ a1 := d1 \]

\[ a1.copy(...) \]
```
Removing the ambiguity

class D
inherit LIST [ T ]
end

select
  copy,
  is_equal
end

C
rename
  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.