



# 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

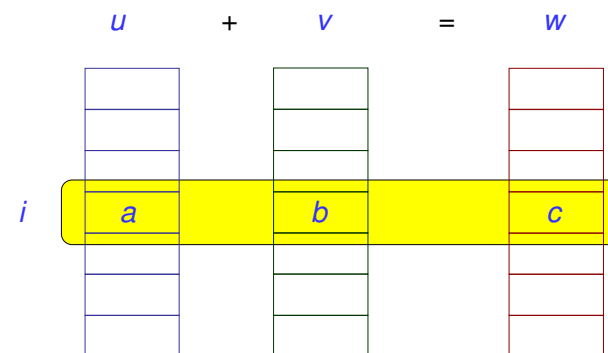


## Agenda for today

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



## Adding two vectors



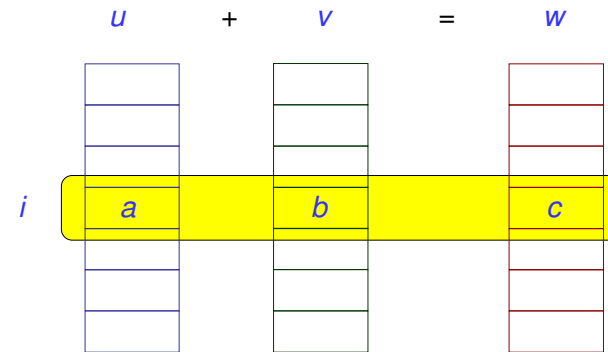
## Constrained genericity

5

```
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

7



## Constrained genericity

6

- The body of **infix** "+":

```
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

8

- 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

9

- 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

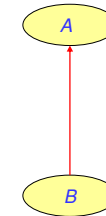
11

- 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

10

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

## Agenda for today

12

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

## Once routines

13

- If instead of

```
r is
  do
    ... Instructions ...
  end
```

- you write

```
r is
  once
    ... Instructions ...
  end
```

- 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

15

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

## Scheme for shared objects

14

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

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

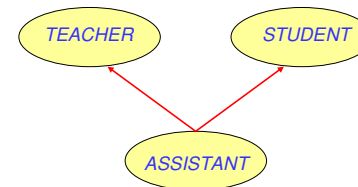
.....

class
  MY_APPLICATION_CLASS
inherit
  SHARED_OBJECTS
feature
  r is
    do
      error_window.put (my_error_message)
    end
end
```

## Multiple inheritance

16

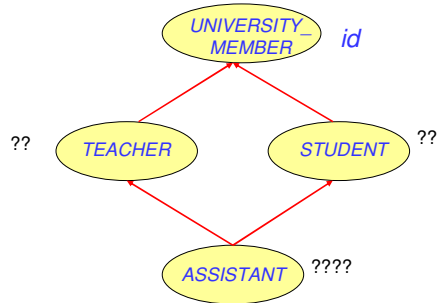
- Allow a class to have two or more parents.
- Examples that come to mind: *ASSISTANT* inherits from *TEACHER* and *STUDENT*.



## Example: Teaching assistant

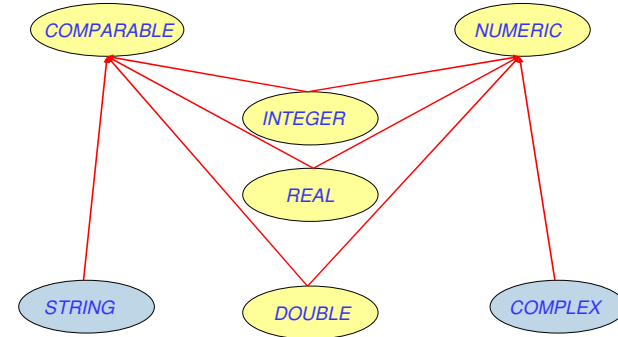
17

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



## Multiple inheritance: Combining abstractions

19



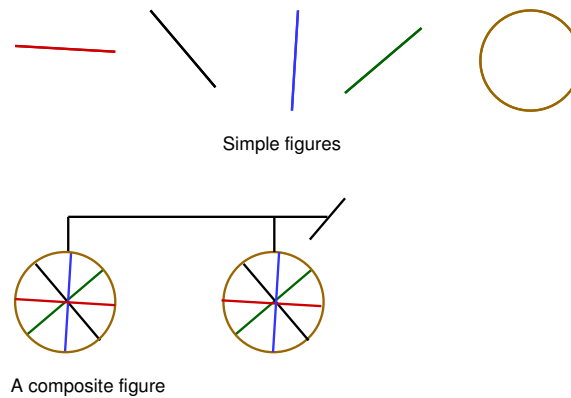
## Other examples of multiple inheritance

18

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

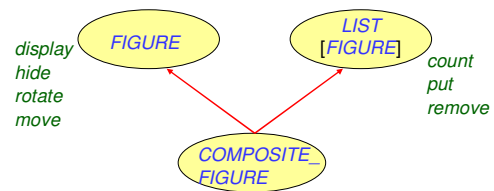
## Multiple inheritance: Composite figures

20



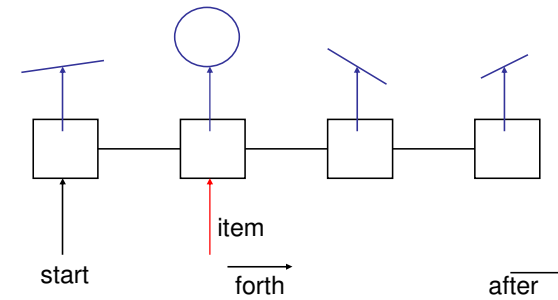
## Defining the notion of composite figure

21



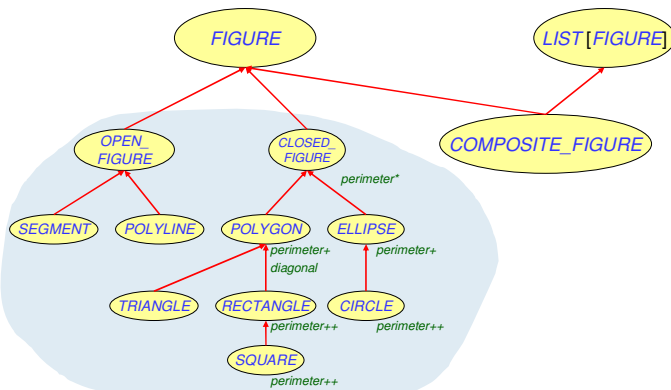
## A composite figure as a list

23



## Composite figures through multiple inheritance

22



## Composite figures

24

```

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

25

- 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.

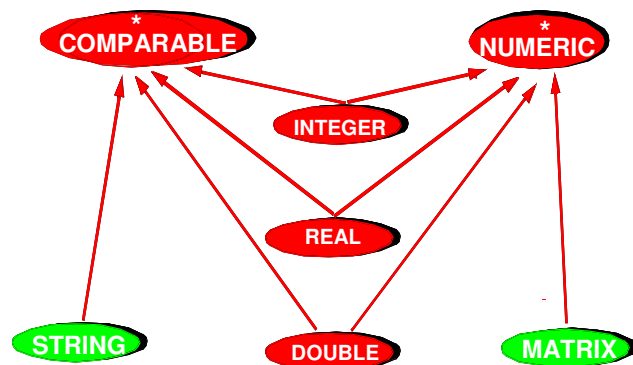
## Multiple inheritance from interfaces: limitations

27

- **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

26



## deferred class COMPARABLE [G] feature

28

```
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 ...
```

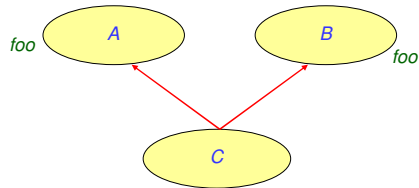
```
infix ">" (other: COMPARABLE [G]) is ...
```

```
...
```

```
end -- class COMPARABLE
```

## Multiple inheritance: Name clashes

29



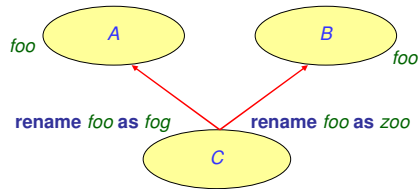
## Resolving name clashes

31

```
class
  C
  inherit
    A
      rename
        foo as fog
      end
    B
      rename
        foo as zoo
      end
  feature
    ...
```

## Resolving name clashes

30



## Results of renaming

32

```
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
```

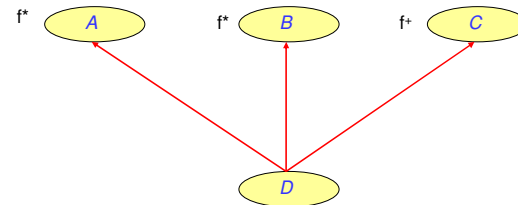
## When is a name clash acceptable?

33

- (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.

## Feature merging

35



## Another application of renaming

34

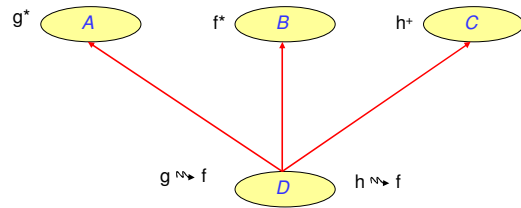
- Provide locally better adapted terminology.
- Example: *child* (*TREE*); *subwindow* (*WINDOW*).

## Feature merging (cont'd)

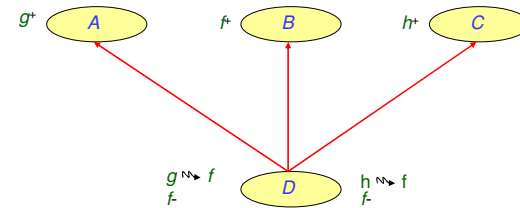
36

```
class D inherit
  A
  B
  C
feature
  ...
end
```

## Feature merging: with different names 37



## Feature merging: effective features 39



*a1: A      b1: B      c1: C      d1: D*  
*a1.g      b1.f      c1.h      d1.f*

## Feature merging: with different names 38

```
class D inherit
  A
  rename
  end
  B
  C
  rename
  end
feature
  ...
end
```

## Feature merging: effective features 40

```
class D inherit
  A
  rename
  undefine
  end
  B
  C
  rename
  undefine
  end
feature
  ...
end
```



End of lecture 8