# Object Oriented History

Akim Demaille akim@lrde.epita.fr
Roland Levillain roland@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

June 14, 2012

# Part I

## Early Languages

1. Simula
   - People Behind SIMULA
   - SIMULA I
   - Simula 67
2. Smalltalk
   - The People Behind Smalltalk
   - Smalltalk 72
   - Smalltalk 76
   - Smalltalk 80
3. C++
   - The Man Behind C++
   - C++

# Simula

# People Behind SIMULA

# Simula


Figure: Ole-Johan Dahl [1]

# Simula

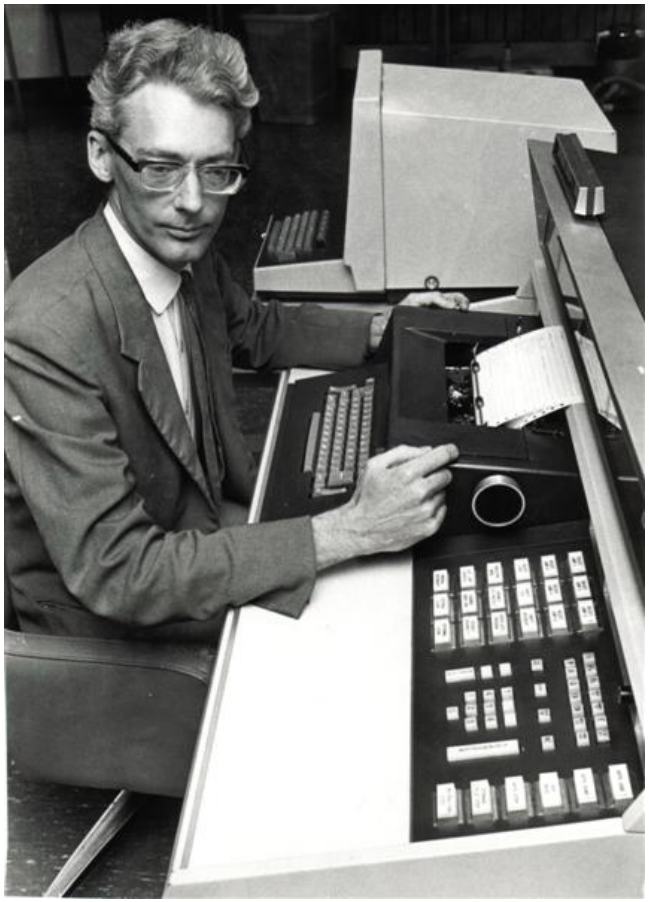


Figure: Ole-Johan Dahl [1]

# Simula

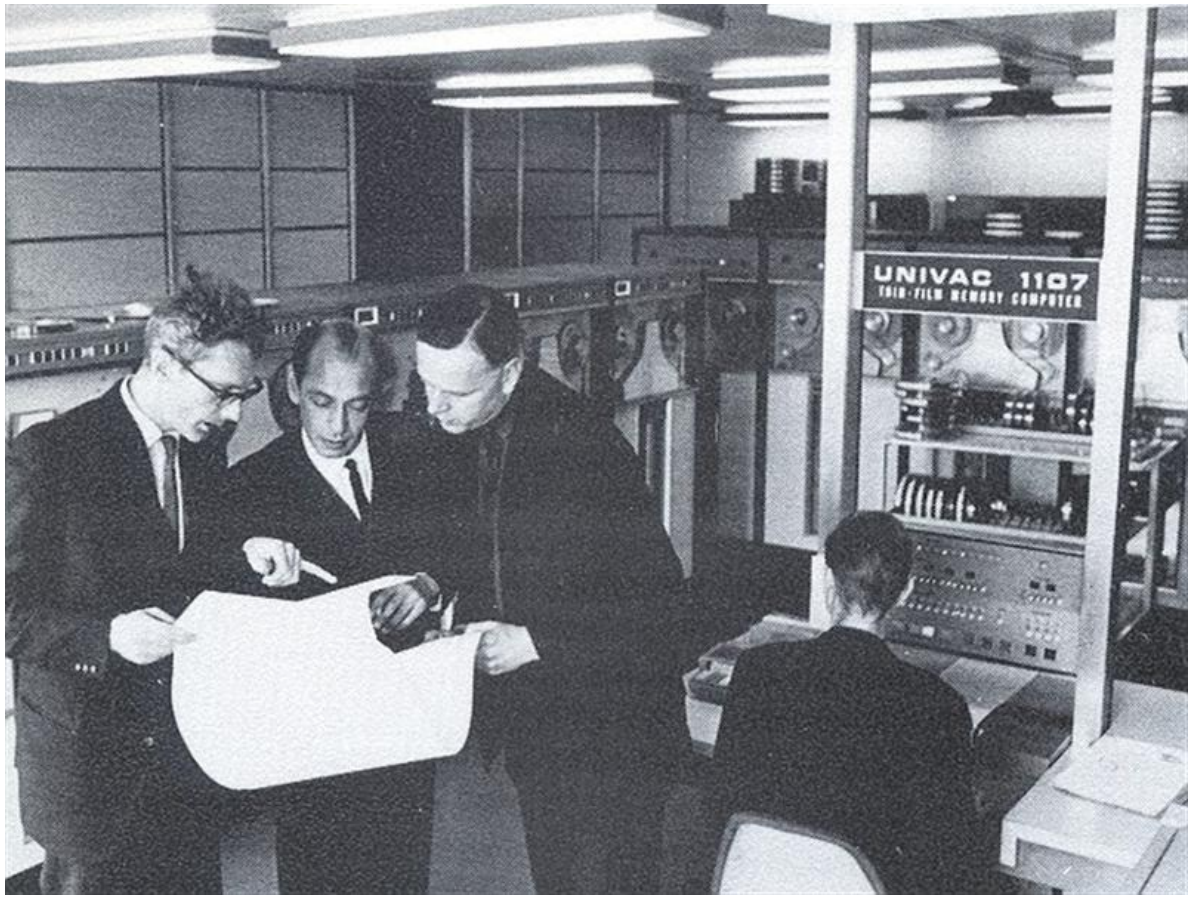



Figure: Dahl & Nygaard [1]

# Simula


Figure: Ole-Johan Dahl & Kristen Nygaard (ca. 1963)

# Simula

Figure: Nygaard & Dahl: Turing Award 2001

# 2002... Sad Year [1]

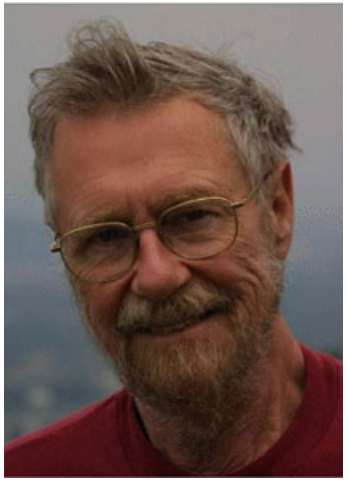
**Ole-Johan Dahl**

Oct 12, 1931, Mandal, NO
June 29, 2002, Asker, NO
"...are there too many basic mechanisms floating around doing nearly the same thing?"

**Kristen Nygaard**

Aug 27, 1926, Oslo, NO
Aug 10, 2002, Oslo, NO
"To program is to understand!"

**Edsger Wybe Dijkstra**



May 11, 1930, Rotterdam, NL
Aug 06, 2002, Nuenen, NL
"Do only what only you can"

# SIMULA I

# Simula

> *In the spring of 1967 a new employee at the NCC in a very shocked voice told the switchboard operator: "two men are fighting violently in front of the blackboard in the upstairs corridor. What shall we do?" The operator came out of her office, listened for a few seconds and then said: "Relax, it's only Dahl and Nygaard discussing SIMULA." — Kristen Nygaard, Ole-Johan Dahl.*

Physical system models. Norwegian nuclear power plant program.

# Simula I

- An Algol 60 preprocessor;
- A subprogram library;
- An original per “process” stack allocation scheme.

Not yet the concept of objects.

# Simula 67

# Simula 67

- Introduces:
  - the concept of object,
  - the concept of class,
  - literal objects (constructors),
  - the concept of inheritance
    (introduced by C. A. R. Hoare for records),
  - the concept of virtual method,
  - attribute hiding!
- Immense funding problems
  steady support from C. A. R. Hoare, N. Wirth and D. Knuth.
- Standardized ISO 1987.

# Shape

```
class Shape(x, y); integer x; integer y;
virtual: procedure draw is procedure draw;;
begin
   comment -- get the x & y components for the object --;
   integer procedure getX;
      getX := x;
   integer procedure getY;
      getY := y;
   comment -- set the x & y coordinates for the object --;
   integer procedure setX(newx); integer newx;
      x := newx;
   integer procedure setY(newy); integer newy;
      y := newy;
   comment -- move the x & y position of the object --;
   procedure moveTo(newx, newy); integer newx; integer newy;
      begin
         setX(newx);
         setY(newy);
      end moveTo;
   procedure rMoveTo(deltax, deltay); integer deltax; integer deltay;
      begin
         setX(deltax + getX);
         setY(deltay + getY);
      end moveTo;
end Shape;
```

# Rectangle

```
Shape class Rectangle(width, height);
  integer width; integer height;
begin
   comment -- get the width & height of the object --;
   integer procedure getWidth;
      getWidth := width;
   integer procedure getHeight;
      getHeight := height;
   comment -- set the width & height of the object --;
   integer procedure setWidth(newwidth); integer newwidth;
      width := newwidth;
   integer procedure setHeight(newheight); integer newheight;
      height := newheight;
   comment -- draw the rectangle --;
   procedure draw;
      begin
         Outtext("Drawing a Rectangle at:(");
         Outint(getX, 0);  Outtext(",");  Outint(getY, 0);
         Outtext("), width ");  Outint(getWidth, 0);
         Outtext(", height ");  Outint(getHeight, 0);
         Outimage;
      end draw;
end Circle;
```

# Circle

```
Shape class Circle(radius); integer radius;
begin
   comment -- get the radius of the object --;
   integer procedure getRadius;
      getRadius := radius;

   comment -- set the radius of the object --;
   integer procedure setRadius(newradius); integer newradius;
      radius := newradius;

   comment -- draw the circle --;
   procedure draw;
      begin
         Outtext("Drawing a Circle at:(");
         Outint(getX, 0);
         Outtext(",");
         Outint(getY, 0);
         Outtext("), radius ");
         Outint(getRadius, 0);
         Outimage;
      end draw;
end Circle;
```

# Top level

```
comment -- declare the variables used --;
ref(Shape) array scribble(1:2);
ref(Rectangle) arectangle;
integer i;

comment -- populate the array with various shape instances --;
scribble(1) :- new Rectangle(10, 20, 5, 6);
scribble(2) :- new Circle(15, 25, 8);

comment -- iterate on the list, handle shapes polymorphically --;
for i := 1 step 1 until 2 do
   begin
      scribble(i).draw;
      scribble(i).rMoveTo(100, 100);
      scribble(i).draw;
   end;

comment -- call a rectangle specific instance --;
arectangle :- new Rectangle(0, 0, 15, 15);
arectangle.draw;
arectangle.setWidth(30);
arectangle.draw;
```

# Execution

```
> cim shape.sim
Compiling shape.sim:
gcc -g -O2 -c shape.c
gcc -g -O2 -o shape shape.o -L/usr/local/lib -lcim
> ./shape
Drawing a Rectangle at:(10,20), width 5, height 6
Drawing a Rectangle at:(110,120), width 5, height 6
Drawing a Circle at:(15,25), radius 8
Drawing a Circle at:(115,125), radius 8
Drawing a Rectangle at:(0,0), width 15, height 15
Drawing a Rectangle at:(0,0), width 30, height 15
```

# Impact of Simula

All the object-oriented languages inherit from Simula.

Smalltalk further with object orientation,
further with dynamic binding.

Objective-C, Pascal, C++, etc.
further with messages.

CLOS further with method selections.

Eiffel further with software engineering,
further with inheritance.

C++ further with static typing and static binding,
deeper in the *.

Hybrid languages logic, functional, assembly, stack based etc.

# Smalltalk

1. Simula
2. Smalltalk
   - The People Behind Smalltalk
   - Smalltalk 72
   - Smalltalk 76
   - Smalltalk 80
3. C++

# Smalltalk

> *We called Smalltalk Smalltalk so that nobody would expect anything from it. — Alan Kay*

Principles:

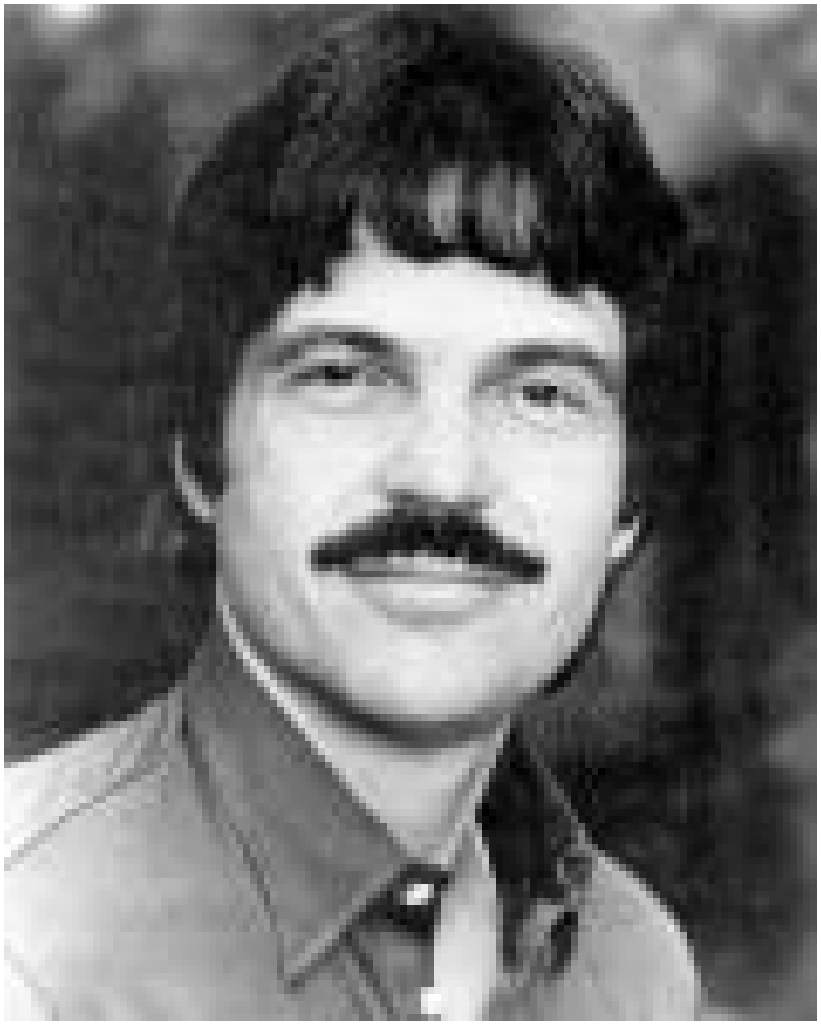
- Everything is object;
- Every object is described by its class (structure, behavior);
- Message passing is the only interface to objects.

Origin:

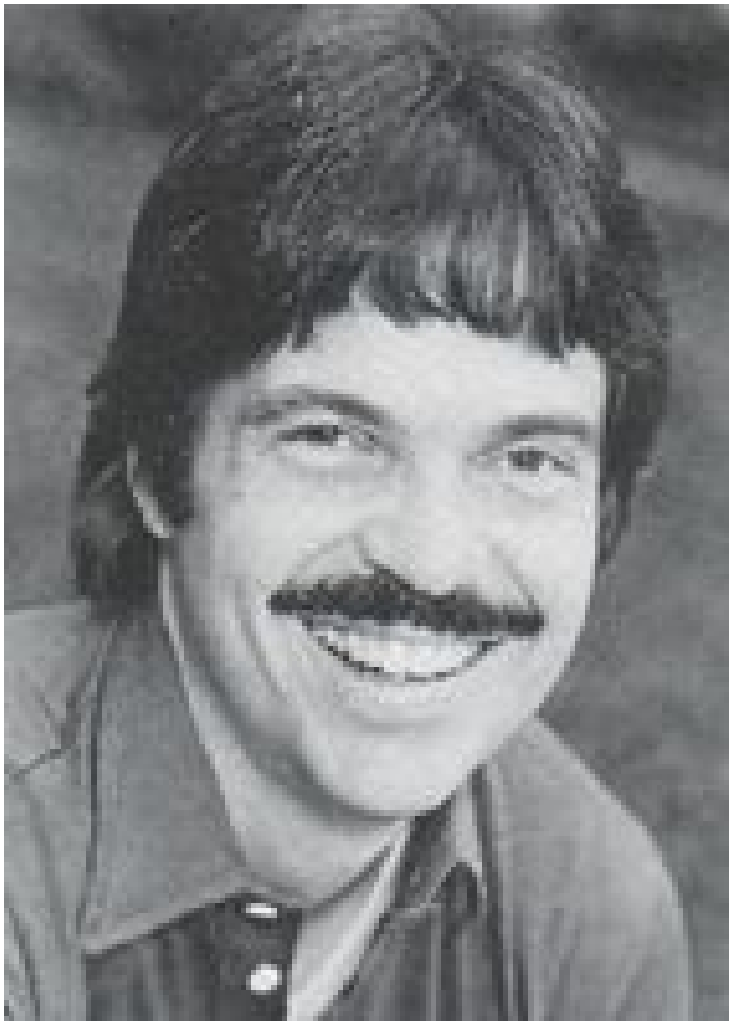
- A programming language that children can understand;
- To create "tomorrow's computer": Dynabook.

# The People Behind Smalltalk

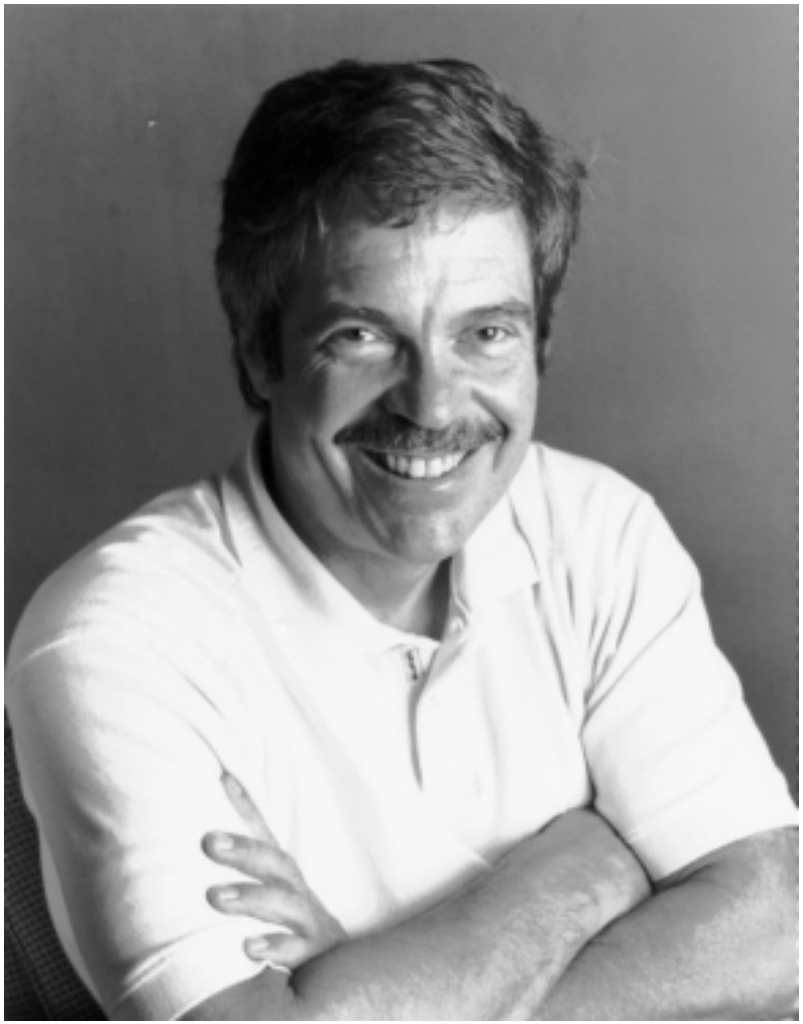
# Alan Kay



# Alan Kay, 1984



# Alan Kay

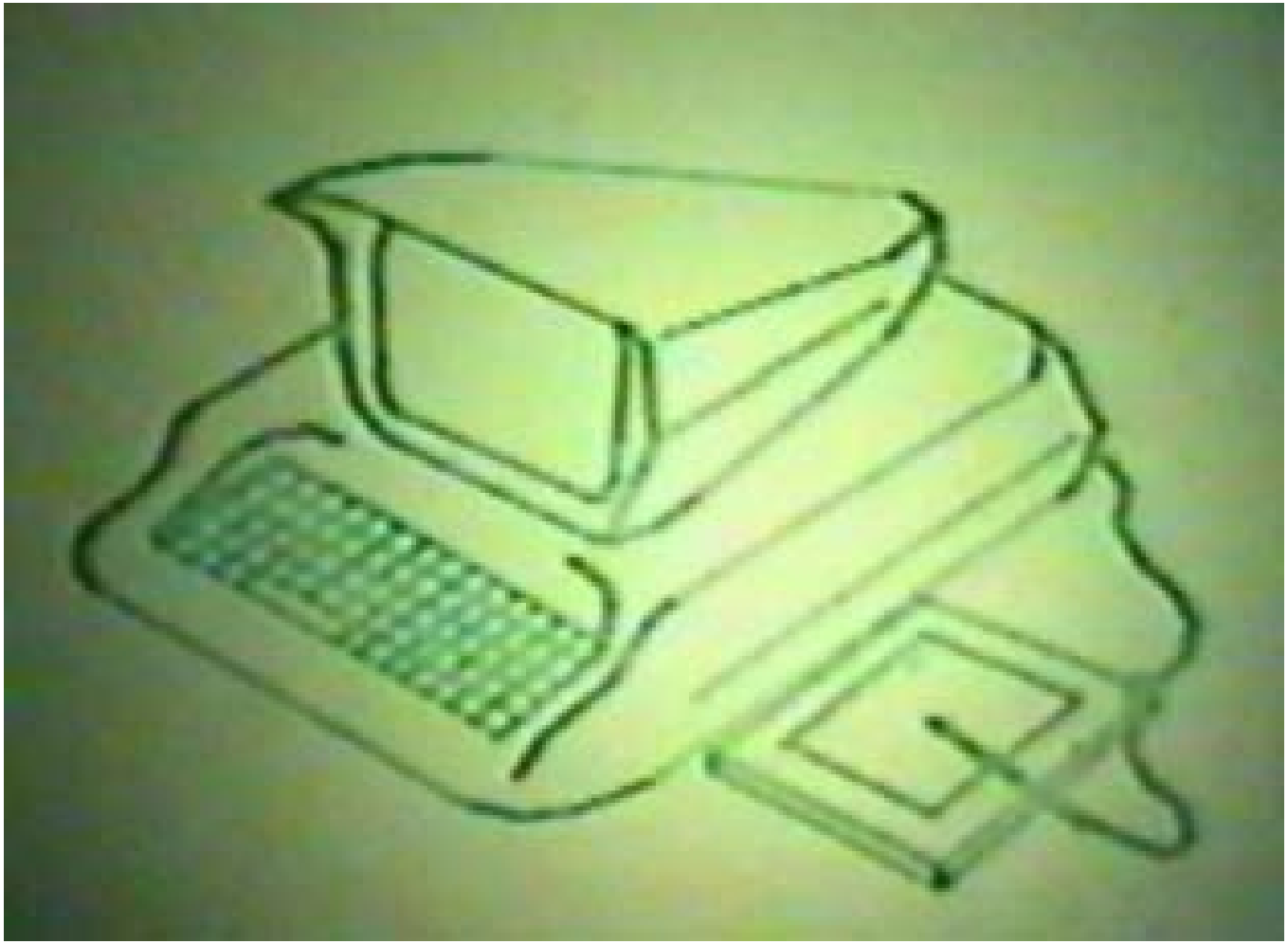


# Ivan Sutherland's Sketchpad 1967 [6]

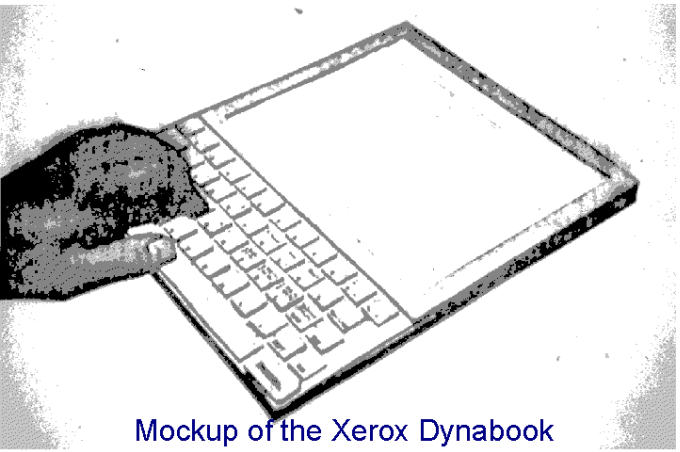
# Douglas Engelbart's NLS 1974


oN Line System [2]

# Flex Machine 1967 [2]



# DynaBook [5]



Mockup of the Xerox Dynabook

*It would have, "enough power to outrace your senses of sight and hearing, enough capacity to store for later retrieval thousands of page-equivalents of reference material, poems, letter, recipes, records, drawings, animations, musical scores, waveforms, dynamic simulations, and anything else you would like to remember and change...". [4]*

To put this project in context, the smallest general purpose computer in the early 1970s was about the size of a desk and the word "multimedia" meant a slide-tape presentation.

# DynaBook [6]

# Smalltalk 72

# Smalltalk 72

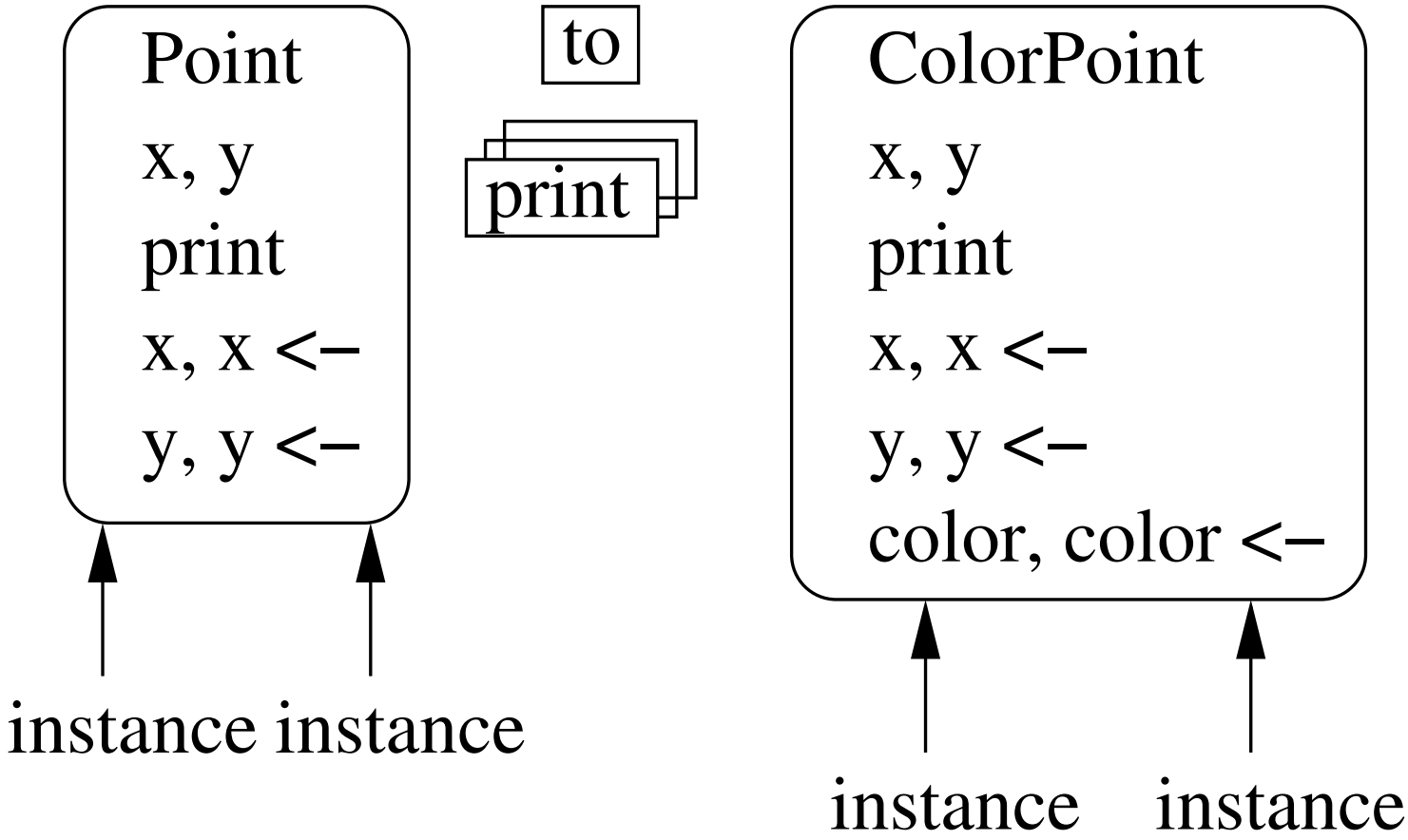
- Written in BASIC.
- Reuses the classes and instances from Simula 67.
- Adds the concept of “message”.
  Dynamic method lookup.

# Smalltalk 72 Sample

```
to Point
  | x y
  (
   isnew => ("x <- :.
     "y <- :.)
   <) x  => ( <) <- => ("x <- : )
      ^ x)
   <) y  => ( <) <- => ("y <- : )
      ^ y)
   <) print => ("( print.
x print.
", print.
y print.
") print.)
  )
=> Point
```

```
center <- Point 0 0
=> (0,0)
center x <- 3
=> (3,0)
center x print
=> 3
```

# Classes and Instances in Smalltalk 72



# Smalltalk 72 Criticisms

- `to` is a primitive, not a method.
- A class is not an object.
- The programmer implements the method lookup.
- Method lookup is too slow.
- No inheritance.

$\Rightarrow$ Programmers were using global procedures.
But some successes:

- Pygmalion
  "Programming by examples"
  inspired Star.

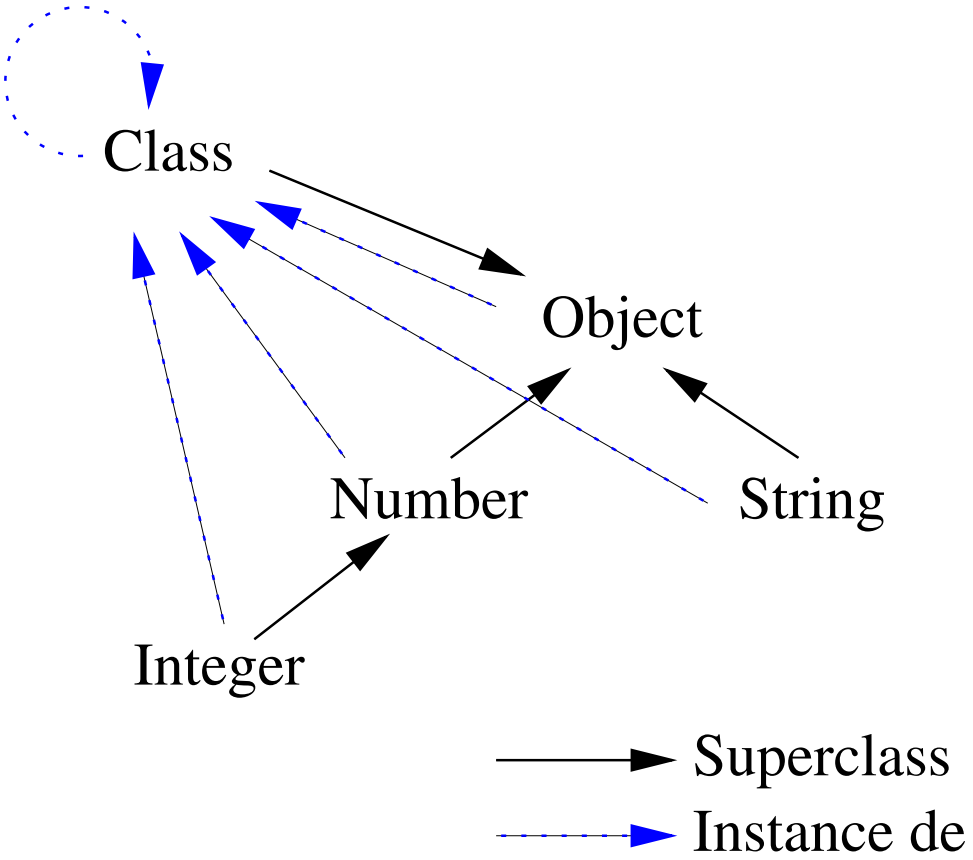
# Smalltalk 76

# Smalltalk 76

- Introduction of the `Class` class.
  The class of classes. Instance of itself. *Metaclass*. How to print a class, add method, instantiate etc.
- Introduction of the `Object` class.
  Default behavior, shared between all the objects.
- Introduction of dictionaries.
  Message handling is no longer handled by the programmers.
- Introduction of inheritance.
- Removal of the `to` primitive.
  Replaced by the `new` message sent to `Class`:

```
Class new title: 'Rectangle';
 fields: 'origin corner'.
```

# Instantiation, inheritance in Smalltalk 76




- Objects keep a link with their generator: `is-instance-of`

# Smalltalk 76 Criticism

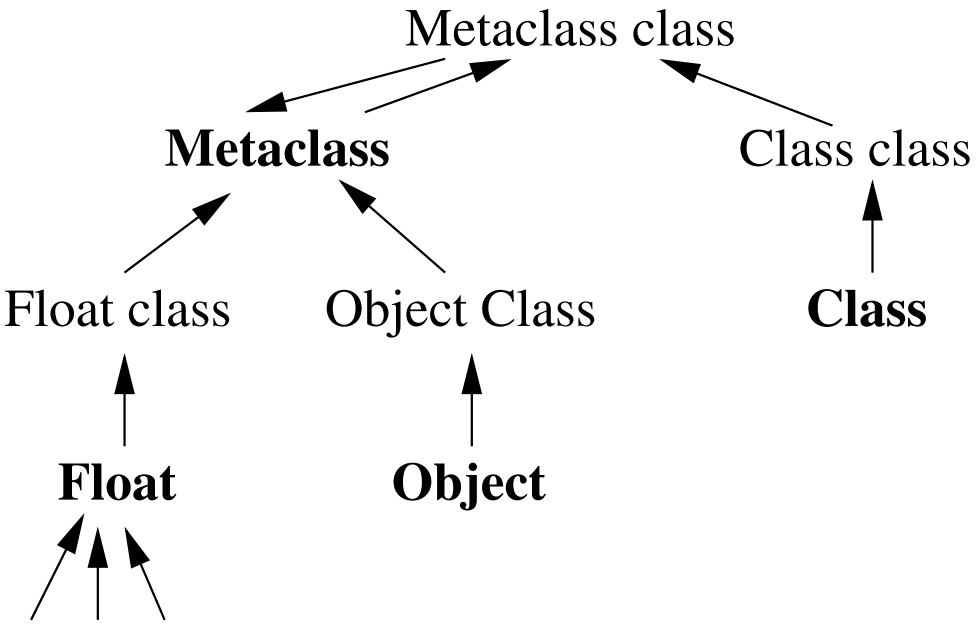
- Significant improvement:
  - Byte-code and a virtual machine provide a 4-100 speedup.
  - *ThingLab*, constraint system experimentation.
  - *PIE*, *Personal Information Environment*.
- But:
  - A single metaclass
    hence a single behavior for classes
    (no specific constructors, etc.).

# Smalltalk 80

# Smalltalk 80

- Deep impact over computer science of the 80's.
- Most constructors take part
  (Apple, Apollo, DEC, HP, Tektronix...).
- Generalization of the metaclass concept.

# Is-instance-of in Smalltalk 80



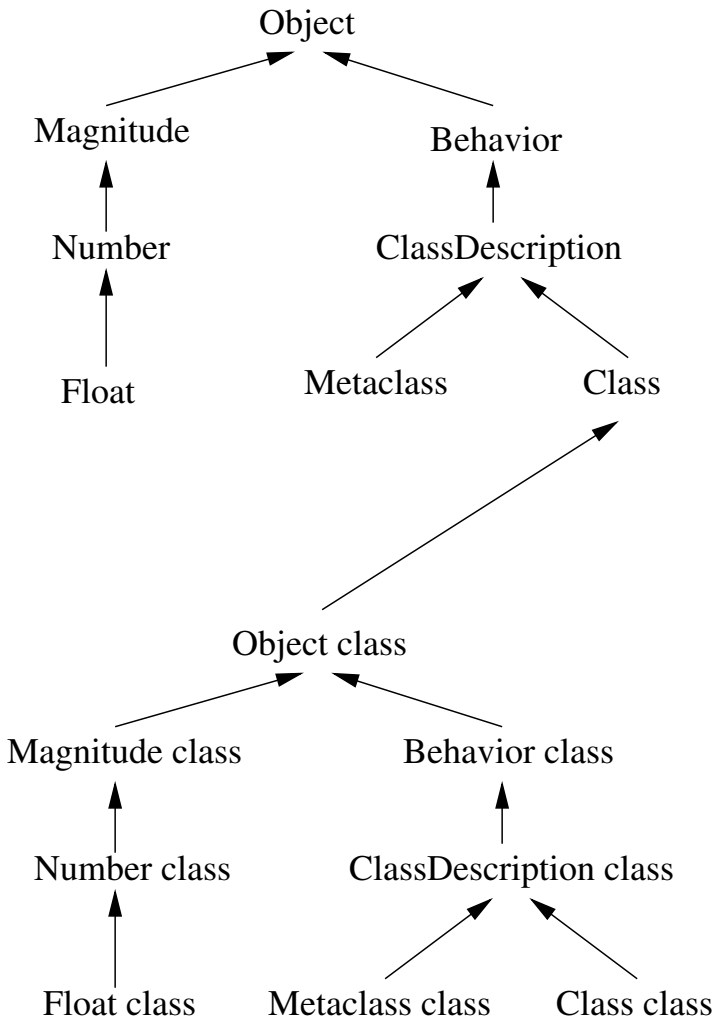

Three layer model:

**Metaclass.** Class behavior (instantiation, initialization, etc.).

**Class.** Type and behavior of objects.

**Instances.** The objects.

# Inheritance in Smalltalk 80



# The Smalltalk 80 System

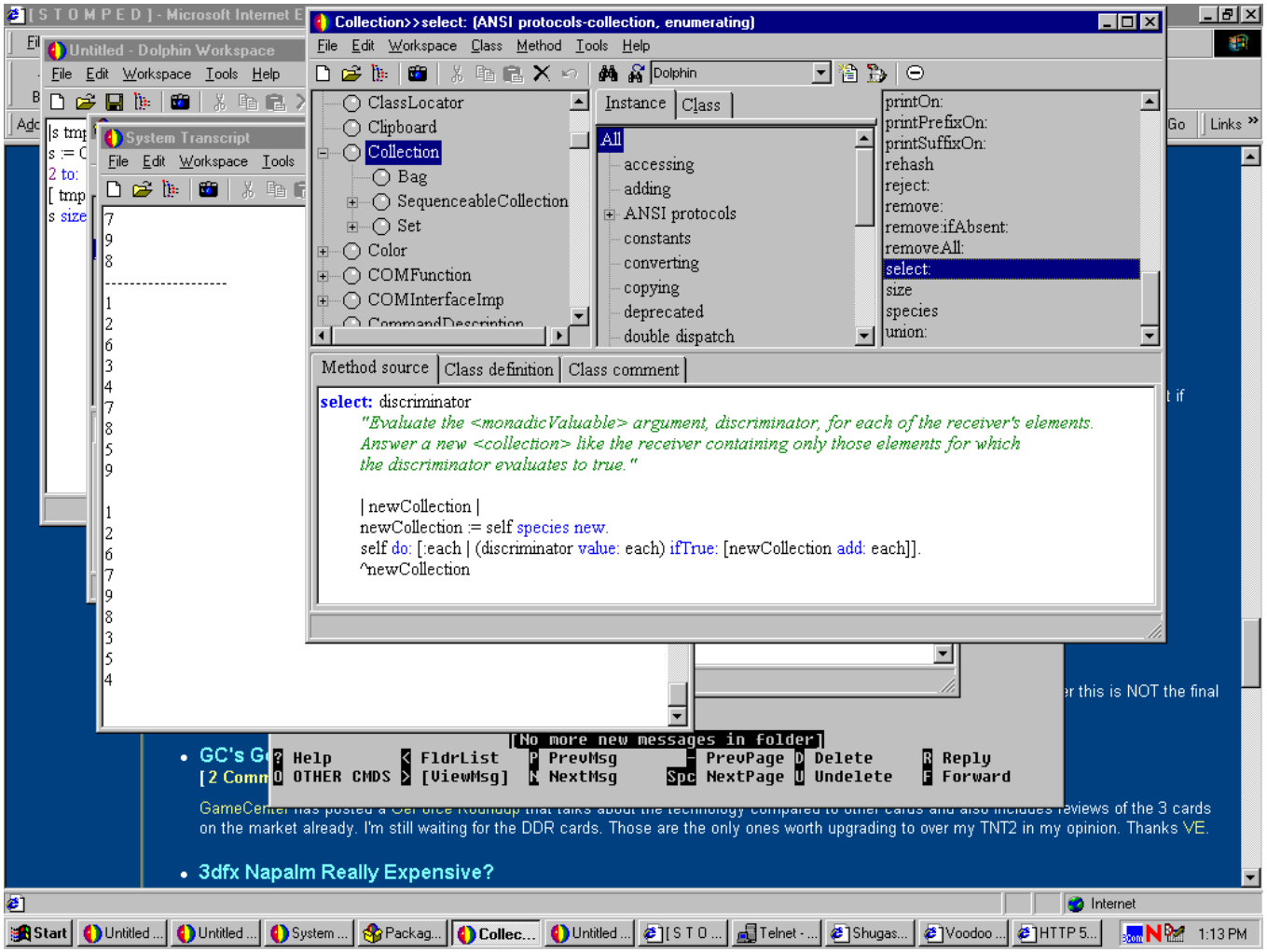
More than a language, a system where *everything* is an object, and the only control structure is message passing.

- a *virtual image*;
- a byte-code compiler;
- a virtual machine;
- more than 500 classes, 4000 methods, 15000 objects.

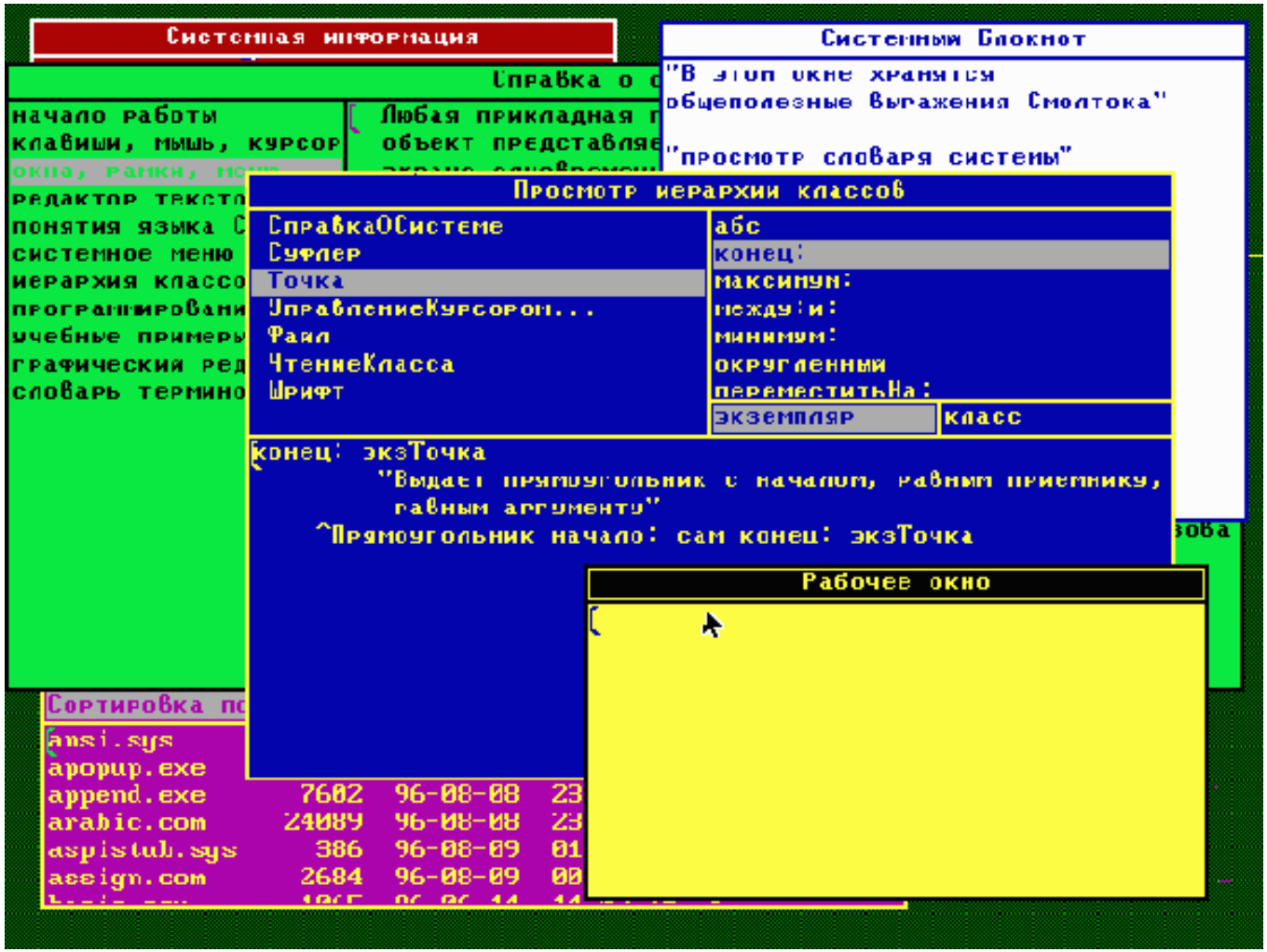
# Smalltalk 80 Standard Library

- System
  `Class`, `Object`, `Number`, `Boolean`, `BlockContext` etc.
- Programming Environment
  `Model`, `View`, `Controler`, etc.
- Standard Library
  `Collection`, `Stream`, etc.
- Notable inventions
  `Bitmap`, `Mouse`, `Semaphore`, `Process`, `ProcessScheduler`

# Smalltalk 80



# Smalltalk 80



# Booleans in Smalltalk 80

- An abstract superclass: `Boolean`
- Two concrete subclasses: `True`, `False`
- Two pre-instantiated objects: `true`, `false`

# Booleans: Logical Operators

Boolean methods: `and:`, `or:`, `not:`.

- In the `True` class

```
and: aBlock
"Evaluate aBlock"
↑ aBlock value
```

- In the `False` class

```
and: aBlock
"Return receiver"
↑ self
```

# Booleans: Control Structures

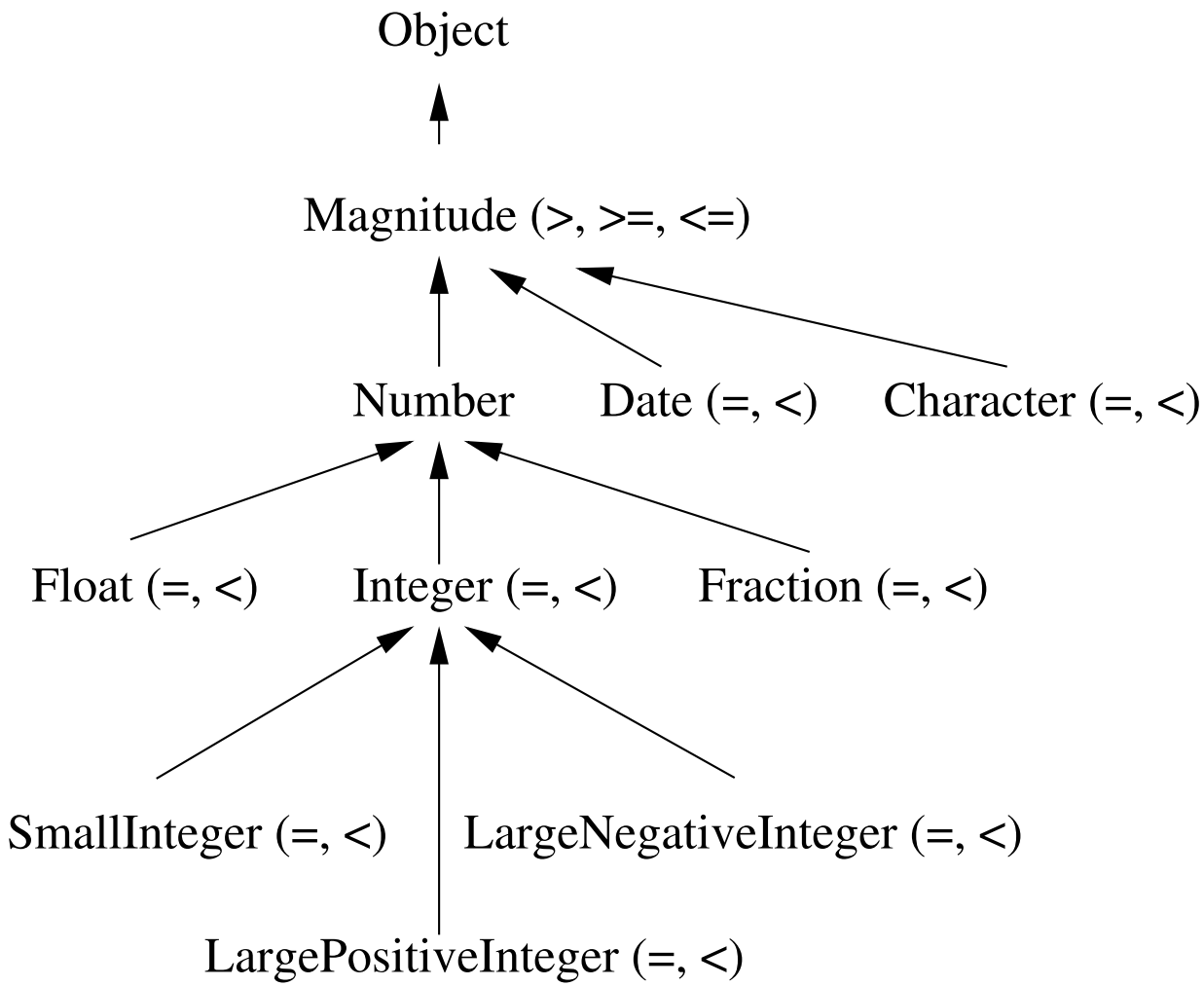
More Boolean methods:

- `ifTrue:`
- `ifFalse:`
- `ifTrue:ifFalse:`
- ... `ifFalse:ifTrue:`

For instance, compute a minimum:

```
| a b x |
...
a <= b ifTrue:  [ x <- a ]
       ifFalse: [ x <- b ].
...
```

# Integers in Smalltalk 80



# Integers in Smalltalk 80

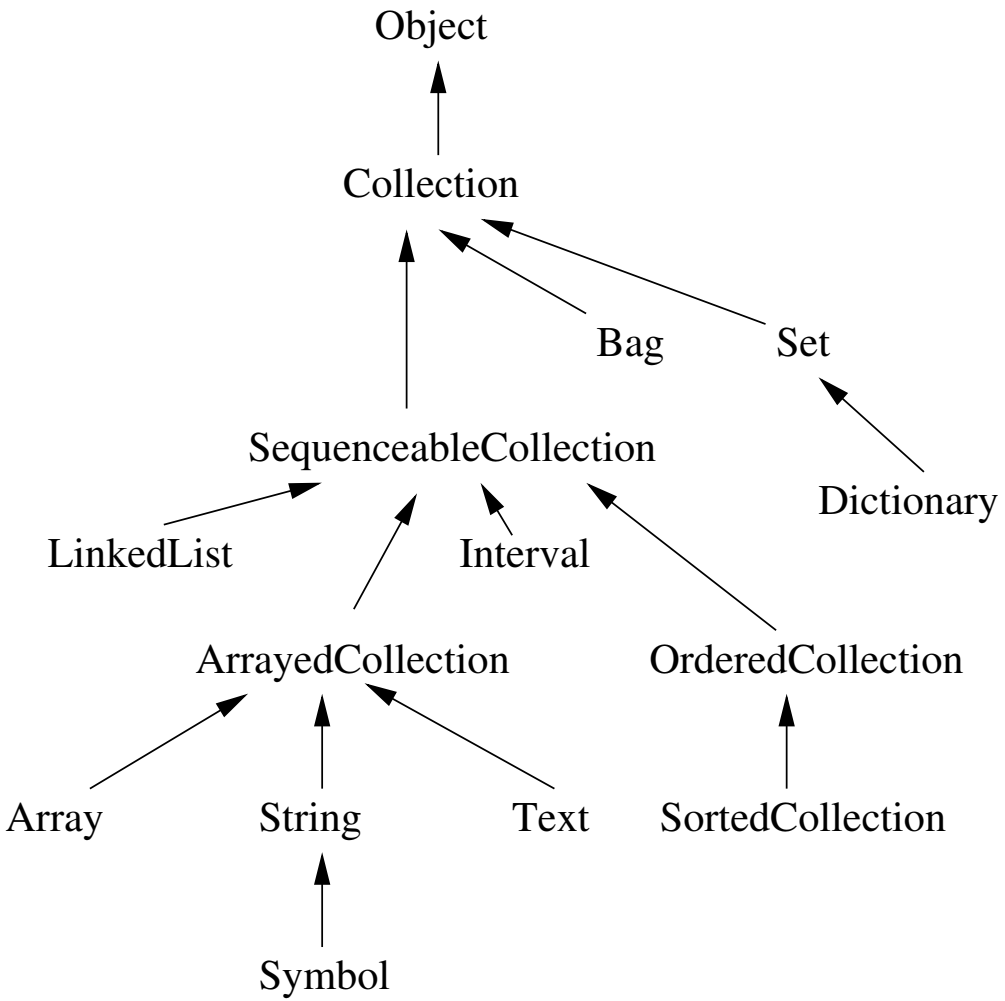
## In Magnitude

```
>= aMagnitude
  ↑ (self < aMagnitude) not
```

## In Date

```
< aDate
  year < aDate year
    ifTrue: [↑ day < aDate day]
   ifFalse: [↑ year < aDate year]
```

# Collections in Smalltalk 80



# Collections in Smalltalk 80

In `LinkedList`:

```
do: aBlock
  | aLink |
  aLink <- firstLink.
  [aLink = nil] whileFalse:
  [aBlock value: aLink.
   aLink <- aLink nextLink]
```

# Using Smalltalk 80 Collections

```
sum <- 0.
#(2 3 5 7 11) do:
      [ :prime |
     sum <- sum + (prime * prime) ]
```

or:

```
sum <- 0.
#(2 3 5 7 11)
      collect: [ :prime | prime * prime ];
   do: [ :number | sum <- sum + number ]
```

# The Smalltalk 80 Environment

- Everything is sorted, classified,
  so that the programmers can browse the system.
- Everything is object.
- The system is reflexive.
- The *inspector* to examine an object.
- Coupled to the debugger and the interpretor,
  a wonderful programming environment.
- Big success of Smalltalk in prototyping.

# Sub-classing in Smalltalk 80: Complexes

- Chose a superclass: `Number`.
- Browse onto it (look in the `Numeric-Numbers` *category*). A skeleton is proposed.

```
Number subclass: #Complex
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Numeric-Numbers'
```

- Complete.

```
Number subclass: #Complex
  instanceVariableNames: 're im'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Numeric-Numbers'
```

# Sub-classing in Smalltalk 80: Complexes

- Validate.
- Go into the `Complex` class, class methods, and create:

```
re: realPart im: imPart
  ↑ (self new) setRe: realPart setIm: imPart
```

# Sub-classing in Smalltalk 80: Complexes

Instance methods:

```
setRe: realPart setIm: imPart
  re <- realPart.
  im <- imPart
```

```
im
  "Return the imaginary part of the receiver."
  ↑ im
```

```
+ aComplex
  ↑ Complex re: (re + aComplex re)
    im: (im + aComplex im)
```

But then:

```
(Complex re: 42 im: 51) + 666
```

yields `message not understood: re`.

# Sub-classing in Smalltalk 80: Complexes

First solution: implement `asComplex` in `Number` and `Complex`

```
"Class Number: addition."
+ aNumber
   | c |
   c <- aNumber asComplex.
   ↑ Complex re: (re + c re) im: (im + c im)
```

Second solution: implement `re` and `im` in `Number`.

But these don't address:

```
666 + (Complex re: 42 im: 51)
```

This issue was known by Smalltalk designers who faced it for other `Number` subclasses; they introduced the `generality` class method.

# Smalltalk 80 Criticism

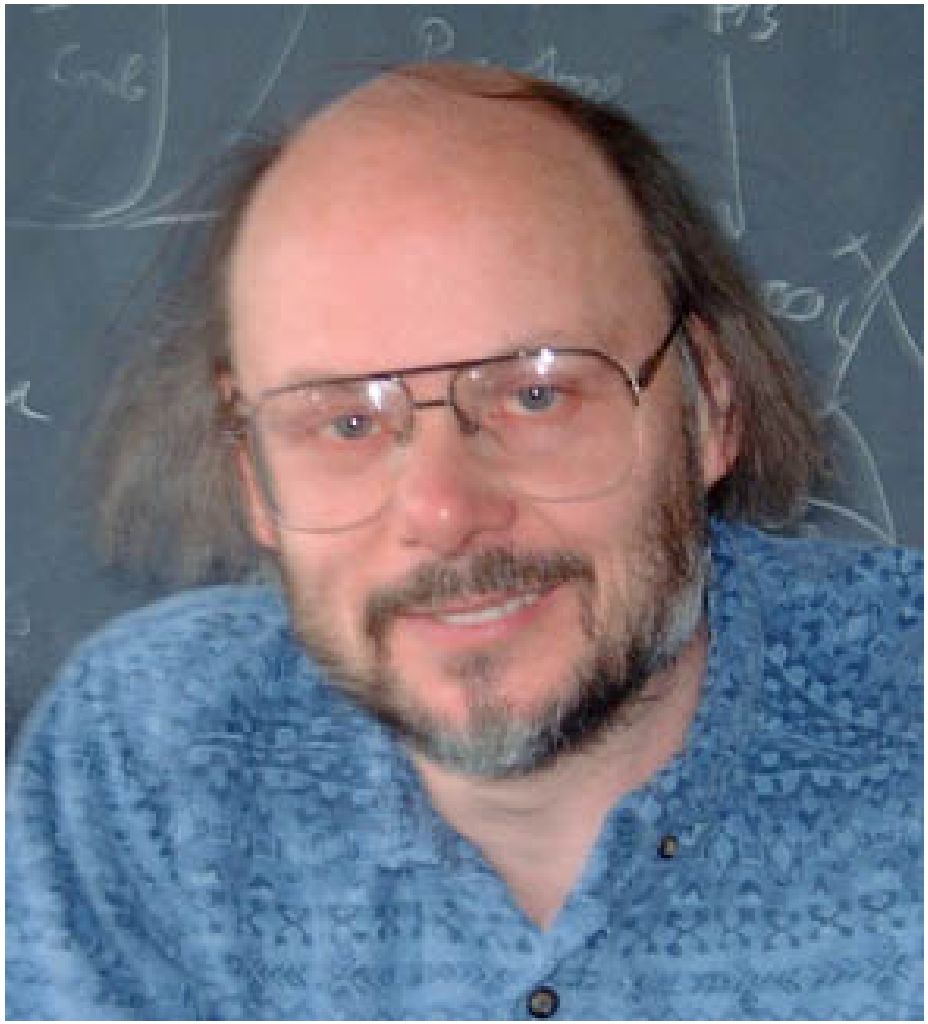
- Some loopholes in the semantics.
- The metaclass concept was considered too difficult.
- No typing!
- Dynamic dispatch exclusively, that's slow.
- The GC is nice, but slow too.
- The virtual image prevents collaborative development.
- No security (one can change *anything*).
- No means to produce standalone applications.
- No multiple inheritance.

# C++

# The Man Behind C++

# Bjarne Stroustrup

# Bjarne Stroustrup



# Bjarne Stroustrup

# Young Bjarne Stroustrup

# Bjarne Stroustrup 2008

# C++

# C++

- Bjarne Stroustrup, BellLabs, 1982.
- `cfront`, a C preprocessor.
- G++, the first real C++ compiler.
- Standardized in 1995.

# C++: A better & safer C

- introduction of `const`;
- introduction of reference;
- introduction of prototypes;
- introduction of Booleans;
- declaring variable anywhere;
- introduction of `void`;
- introduction of `inline`;
- introduction of `namespace`;
- introduction of overloading etc.

Most features made it into more moderns C.

# Class declaration

```
#ifndef SHAPE_HH_
# define SHAPE_HH_ 1

class Shape
{
public:
  Shape (int x, int y);

  void x_set (int x); int x_get () const;
  void y_set (int y); int y_get () const;

  void move_to (int x, int y);
  void rmove_to (int deltax, int deltay);

  virtual void draw () const;
private:
  int x_, y_;
};

#endif SHAPE_HH_
```

# Class implementation

```
#include "shape.hh"

// Constructor.
Shape::Shape (int x, int y) : x_ (x), y_ (y) {}

// Accessors for x & y.
int Shape::x_get () const { return x_; }
int Shape::y_get () const { return y_; }
void Shape::x_set (int x) { x_ = x; }
void Shape::y_set (int y) { y_ = y; }

// Move the shape.
void Shape::move_to (int x, int y) { x_set (x); y_set (y); }
void Shape::rmove_to (int x, int y) {
  move_to (x_get () + x, y_get () + y);
}

// Abstract draw method.
void Shape::draw () const { abort (); }
```

# Class definition

```
#ifndef SHAPES_HH_
# define SHAPES_HH_ 1
class Shape
{
public:
  Shape (int x, int y) : x_ (x), y_ (y) {}

  int x_get () const { return x_; }
  int y_get () const { return y_; }

  void x_set (int x) { x_ = x; }
  void y_set (int y) { y_ = y; }

  void move_to (int x, int y) { x_ = x; y_ = y; }
  void rmove_to (int x, int y) { x_ += x; y_ += y; }
  virtual void draw () const = 0;
private:
  int x_, y_;
};
#endif // ! SHAPES_HH_
```

# Sub-classing: rectangle.hh

```
#ifndef RECTANGLE_HH_
# define RECTANGLE_HH_ 1
# include <iostream>
# include "shape.hh"
class Rectangle: public Shape {
public:
  Rectangle (int x, int y, int w, int h) :
    Shape (x, y), width_ (w), height_ (h) {}
  int width_get  () const { return width_; }
  int height_get () const { return height_; }
  int width_set  (int w)  { width_ = w; }
  int height_set (int h)  { height_ = h; }
  void draw () const {
    std::cout << "Drawing a Rectangle at: (" << x_get ()
              << "," << y_get () << "), " << "width " << width_
              << ", height " << height_ << std::endl;
  }
private:
  int width_, height_;
};
#endif // ! RECTANGLE_HH_
```

# Sub-classing: circle.hh

```
#ifndef CIRCLE_HH_
# define CIRCLE_HH_ 1
# include <iostream>
# include "shape.hh"
class Circle: public Shape {
public:
  Circle (int x, int y, int r) :
    Shape (x, y), radius_ (r) {}

  int radius_get () const  { return radius_; }
  void radius_set (int r) { radius_ = r; }

  void draw () const {
    cout << "Drawing a Circle at: ("
         << x_get () << "," << y_get () << "), "
         << "radius " << radius_ << endl;
  }
private:
  int radius_;
};
#endif // ! CIRCLE_HH_
```

# Polymorphism

```
#include "shape.hh"
#include "circle.hh"
#include "rectangle.hh"

int
main ()
{
  Shape *shapes[2];
  shapes[0] = new Rectangle (10, 20, 5, 6);
  shapes[1] = new Circle (15, 25, 8);

  for (int i = 0; i < 2; i++)
    {
      shapes[i]->draw ();
      shapes[i]->rmove_to (100, 100);
      shapes[i]->draw ();
    }
}
```

# Polymorphism

Result:

```
Drawing a Rectangle at: (10,20), width 5, height 6
Drawing a Rectangle at: (110,120), width 5, height 6
Drawing a Circle at: (15,25), radius 8
Drawing a Circle at: (115,125), radius 8
```

# Parameterized Polymorphism

```
template <typename T>
T
id (T t)
{
  return t;
}

int
main ()
{
  id (3);
  id (3.0);
  id ("three");
  int three[3] = { 3, 3, 3 };
  id (three);
  id (main);
}
```

# Generic Classes

```
#include <iostream>
template <typename T> struct Pair {
  Pair (T fst, T snd): fst_ (fst), snd_ (snd) {}
  T fst () const { return fst_; }
  T snd () const { return snd_; }
private:
  T fst_, snd_;
};

int main () {
  Pair<int> foo (2, 3);
  std::cout << foo.fst () << ", " << foo.snd () << std::endl;

  Pair<float> bar (2.2, 3.3);
  std::cout << bar.fst () << ", " << bar.snd () << std::endl;

  Pair <Pair<int> > baz (Pair<int> (1, 2), Pair<int> (3, 4));
  std::cout << baz.fst (). fst () << baz.fst ().snd ()
            << baz.snd (). fst () << baz.snd ().snd ()
            << std::endl;
}
```

# Standard Template Library

```
#include <iostream>
#include <iterator>
#include <list>

int
main ()
{
  std::list<int> list;
  list.push_back (1);
  list.push_back (2);
  list.push_back (3);
  std::copy (list.begin (), list.end (),
             std::ostream_iterator <int> (std::cout, "\n"));
}
```

# Quickly Read Only

```
template <class _Tp, class _Alloc, size_t __bufsize>
template <class _ForwardIterator>
void
deque<_Tp,_Alloc,__bufsize>::
    insert(iterator __pos, _ForwardIterator __first,
           _ForwardIterator __last, forward_iterator_tag) {
  size_type __n = 0;
  distance(__first, __last, __n);
  if (__pos._M_cur == _M_start._M_cur) {
    iterator __new_start = _M_reserve_elements_at_front(__n);
    __STL_TRY {
      uninitialized_copy(__first, __last, __new_start);
      _M_start = __new_start;
    }
    __STL_UNWIND(_M_destroy_nodes(__new_start._M_node,
                                  _M_start._M_node));
  }
  else if (__pos._M_cur == _M_finish._M_cur) {
    iterator __new_finish = _M_reserve_elements_at_back(__n);
    __STL_TRY {
      uninitialized_copy(__first, __last, _M_finish);
      _M_finish = __new_finish;
    }
    __STL_UNWIND(_M_destroy_nodes(_M_finish._M_node + 1,
                                  __new_finish._M_node + 1));
  }
```

# Poor Error Messages

```
#include <iostream>
#include <list>

int
main ()
{
  std::list<int> list;
  list.push_back (1);
  list.push_back (2);
  list.push_back (3);
  const std::list<int> list2 = list;

  for (std::list<int>::iterator i = list2.begin ();
       i != list2.end (); ++i)
    std::cout << *i << std::endl;
}
```

# Poor Error Messages

G++ 2.95:

```
bar.cc: In function ‘int main()’:
bar.cc:13: conversion from
   ‘_List_iterator<int,const int &, const int *>’
   to non-scalar type
   ‘_List_iterator<int,int &, int *>’ requested
bar.cc:14: no match for
   ‘_List_iterator<int,int &,int *> & !=
      _List_iterator<int,const int &,const int *>’
/usr/lib/gcc-lib/i386-linux/2.95.4/../../../../include/g++-3/stl_list.h:70:
   candidates are:
   bool _List_iterator<int,int &,int *>::operator !=
     (const _List_iterator<int,int &,int *> &) const
```

# (A Bit Less) Poor Error Messages

Some progress: G++ 3.3.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13: error: conversion from
   ‘std::_List_iterator<int, const int&, const int*>’
   to non-scalar type
   ‘std::_List_iterator<int, int&, int*>’ requested
```

G++ 3.4, 4.0 and 4.1, 4.2, 4.3 and 4.4.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13: error: conversion from
   ‘std::_List_const_iterator<int>’ to non-scalar type
   ‘std::_List_iterator<int>’ requested
```

G++ 4.5.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13:50: error: conversion from
   ‘std::list<int>::const_iterator’ to non-scalar type
   ‘std::list<int>::iterator’ requested
```

# (A Bit Less) Poor Error Messages

G++ 4.6 and 4.7.

```
list-invalid.cc: In function ‘int main()’:
list-invalid.cc:13:50: erreur: conversion from
   ‘std::list<int>::const_iterator {aka std::_List_const_iterator<int>}’
   to non-scalar type
   ‘std::list<int>::iterator {aka std::_List_iterator<int>}’ requested
```

G++ 4.8 (forthcoming).

```
list-invalid.cc: In function ’int main()’:
list-invalid.cc:13:50: error: conversion from
  ’std::list<int>::const_iterator {aka std::_List_const_iterator<int>}’
  to non-scalar type
  ’std::list<int>::iterator {aka std::_List_iterator<int>}’ requested
   for (std::list<int>::iterator i = list2.begin ();
                                                  ^
```

# (A Bit Less) Poor Error Messages

## ICC 8.1 and 9.1.

```
list-invalid.cc(8):
   remark #383: value copied to temporary, reference
                to temporary used
   list.push_back (1);
                   ^

[...]
list-invalid.cc(13): error: no suitable user-defined conversion
   from
   "std::list<int, std::allocator<int>>::const_iterator" to
   "std::list<int, std::allocator<int>>::iterator" exists
    for (std::list<int>::iterator i = list2.begin ();
                                      ^
```

## ICC 10.0 and 11.0.

```
list-invalid.cc(13): error: no suitable user-defined conversion
  from "std::_List_const_iterator<int>"
  to "std::_List_iterator<int>" exists
    for (std::list<int>::iterator i = list2.begin ();
                                      ^
```

# (A Bit Less) Poor Error Messages

## Clang 1.1 (LLVM 2.7)

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                              ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:113:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'struct std::_List_iterator<int> const' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

# (A Bit Less) Poor Error Messages

Clang 2.8 (LLVM 2.8).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                               ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::_List_iterator<int> const &' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

# (A Bit Less) Poor Error Messages

Clang 2.9 (LLVM 2.9).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                               ^   ~~~~~~~~~~~~~~
In file included from list-invalid.cc:2:
In file included from /usr/include/c++/4.2.1/list:69:
/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'const std::_List_iterator<int> &' for 1st argument
    struct _List_iterator
           ^
1 error generated.
```

# (A Bit Less) Poor Error Messages

Clang 3.0 (LLVM 3.0) and Clang 3.1 (LLVM 3.1).

```
list-invalid.cc:13:33: error: no viable conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'std::list<int>::iterator' (aka '_List_iterator<int>')
  for (std::list<int>::iterator i = list2.begin ();
                               ^   ~~~~~~~~~~~~~~


/usr/include/c++/4.2.1/bits/stl_list.h:112:12: note: candidate
      constructor (the implicit copy constructor) not viable:
      no known conversion from
      'const_iterator' (aka '_List_const_iterator<int>') to
      'const std::_List_iterator<int> &' for 1st argument;
    struct _List_iterator
           ^
1 error generated.
```

# Part II

# OO Properties

# Bad Engineering Properties of Object Oriented Languages

# Engineering Properties [3]

- Economy of execution.
  How fast does a program run?
- Economy of compilation.
  How long does it take to go from sources to executables?
- Economy of small-scale development.
  How hard must an individual programmer work?
- Economy of large-scale development.
  How hard must a team of programmers work?
- Economy of language features.
  How hard is it to learn or use a programming language?

# Economy of execution

Type information was first introduced in programming to improve code generation and run-time efficiency for numerical computations. In ML, accurate type information eliminates the need for nil-checking on pointer dereferencing.
Object-oriented style intrinsically less efficient that procedural style (`virtual`). The traditional solution to this problem (analyzing and compiling whole programs) violates modularity and is not applicable to libraries.
Much can be done to improve the efficiency of method invocation by clever program analysis, as well as by language features (e.g. `final`). Design type systems that can statically check many of the conditions that now require dynamic subclass checks.

# Economy of compilation

Type information can be organized into interfaces for program modules (Modula-2, Ada...). Modules can then be compiled independently. Compilation of large systems is made more efficient. The messy aspects of system integration are thus eliminated.
Often, no distinction between the code and the interface of a class. Some object-oriented languages are not sufficiently modular and require recompilation of superclasses when compiling subclasses. Time spent in compilation may grow disproportionally with the size of the system.
We need to adopt languages and type systems that allow the separate compilation of (sub)classes, without resorting to recompilation of superclasses and without relying on “private” information in interfaces.

# Economy of small-scale development

Well designed type systems allow typechecking to capture a large fraction of routine programming errors. Remaining errors are easier to debug: large classes of other errors have been ruled out.
Typechecker as a development tool (changing the name of a type when its invariants change even though the type structure remains the same).
Big win of OO: class libraries and frameworks. But when ambition grows, programmers need to understand the details of those class libraries: more difficult than understanding module libraries. The type systems of most OOL are not expressive enough; programmers must often resort to dynamic checking or to unsafe features, damaging the robustness of their programs.
Improvements in type systems for OOL will improve error detection and the expressiveness of interfaces.

# Economy of large-scale development

Data abstraction and modularization have methodological advantages for development. Negotiate the interfaces, then proceed separately. Polymorphism is important for reusing code modularly.
Teams developing/specializing class libraries. Reuse is a big win of OOL, but poor modularity wrt class extension and modification (method "removal", etc.). Confusion bw classes and object types (limits abstractions). Subtype polymorphism is not good enough for containers.
Formulating and enforcing inheritance interfaces: the contract bw a class and its subclasses. Requires language support development.
Parametric polymorphism is beginning to appear but its interactions with OO features need to be better understood.
Interfaces/subtyping and classes/subclassing must be separated.

# Economy of language features

Well-designed orthogonal constructs can be naturally composed (array of arrays; n-ary functions vs 1-ary and tuples).
Orthogonality reduces the complexity of languages. Learning curve thus reduced, re-learning minimized.
Smalltalk, good. C++ daunting in the complexity of its many features. Somewhere something went wrong; what started as economical and uniform ("everything is an object") ended up as a baroque collection of class varieties. Java represents a healthy reaction, but is more complex than many people realize.
Prototype-based languages tried to reduce the complexity by providing simpler, more composable features, but much remains to be done for class-based languages. How can we design an OOL that allows powerful engineering but also simple and reliable engineering?

# Bibliography I


People behind Informatics — In memory of Ole-Johan Dahl, Edsger Wybe Dijkstra and Kristen Nygaard.
`http://cs-exhibitions.uni-klu.ac.at/`, 2003.

ArtMuseum.
In Depth.
`http://www.artmuseum.net/w2vr/archives/archives.html`, 2003.

Luca Cardelli.
Bad engineering properties of object-oriented languages.
*ACM Computing Surveys (CSUR)*, 28(4es):150, 1996.

# Bibliography II



Alan Kay and Adele Goldberg.
Personal dynamic media.
*Computer*, 31, March 1977.


Mike Sharples.
Why did you bother to write a book?
http://www.eee.bham.ac.uk/sharplem/Routledge/article.htm, 1998.


Alan B. Scrivener.
SIGKIDS 2003 Influences Timeline.
http://san-diego.siggraph.org/sigkids/Influences.html, 2003.